Marcos K. Aguilera (Ed.)

ARCoSS

LNCS 7611

Distributed Computing

26th International Symposium, DISC 2012
Salvador, Brazil, October 2012
Proceedings

Springer

Marcos K. Aguilera (Ed.)

Distributed Computing

26th International Symposium, DISC 2012
Salvador, Brazil, October 16-18, 2012
Proceedings

Springer

Volume Editor

Marcos K. Aguilera
Microsoft Corporation
Building SVC6, 1065 La Avenida
Mountain View, CA 94043, USA
E-mail: marcos_aguilera_msrsvc@live.com

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-33650-8 e-ISBN 978-3-642-33651-5
DOI 10.1007/978-3-642-33651-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012947417

CR Subject Classification (1998): C.2.4, C.2, H.4, D.2, H.3, F.2, I.2.11

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

DISC is the International Symposium on Distributed Computing, an international forum on the theory, design, analysis, implementation, and application of distributed systems and networks. DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS). This volume contains the papers presented at DISC 2012, which was held during 16–18 October 2012 in Salvador, Brazil.

This year, the symposium received 112 regular paper submissions, of which 27 were selected for regular presentations at the symposium. Each regular presentation was accompanied by a paper of up to 15 pages in this volume. The symposium also received 7 brief announcement submissions. Among these submissions and the regular paper submissions, 24 submissions were selected to appear as brief announcements. Each brief announcement reflected ongoing work or recent results, and was accompanied by a two-page abstract in this volume. It is expected that these brief announcements will appear as full papers in other conferences or journals.

Submissions were evaluated in two phases. In the first phase, every submission was evaluated by three members of the program committee. Submissions deemed promising were further examined in the second phase by at least two additional program committee members. As a result of this two-phase review process, every submission was evaluated by at least three program committee members, while every submission accepted for a regular presentation was evaluated by at least five program committee members. Program committee members were assisted by around 122 external reviewers. After the reviews were completed, the program committee engaged in email discussions and made tentative decisions for some papers. The program committee later held a phone meeting on 28 July 2012 to discuss the borderline papers and finalize all decisions.

Revised and expanded versions of several accepted papers will be considered for publication in a special issue of the Distributed Computing journal.

The Best Paper Award of DISC 2012 was presented to Mika Göös and Jukka Suomela for the paper titled "No Sublogarithmic-Time Approximation Scheme for Bipartite Vertex Cover".

The Best Student Paper Award of DISC 2012 was presented to Boris Korenfeld and Adam Morrison for the paper titled "CBTree: A Practical Concurrent Self-Adjusting Search Tree", which was co-authored with Yehuda Afek, Haim Kaplan, and Robert E. Tarjan.

The symposium featured two keynote presentations. The first one was given by Yehuda Afek from Tel-Aviv University, and was titled "Launching Academic Ideas into the Real World". The second keynote presentation was given by Simon Peyton-Jones from Microsoft Research, and was titled "Towards Haskell in the Cloud".

In addition, the symposium included four tutorials. The first tutorial, presented by Elias P. Duarte Jr., was titled “System-Level Diagnosis: A Stroll through 45 Years of Research on Diagnosable Systems”. The second tutorial, presented by Michel Raynal, was titled “Implementing Concurrent Objects in Multiprocessor Machines”. The third tutorial, presented by Nicola Santoro, was titled “An Introduction to Distributed Computing by Mobile Entities: Agents, Robots, Sensors”. The fourth tutorial, presented by Paulo Veríssimo, was titled “Beyond the Glamour of Byzantine Fault Tolerance: OR Why Resisting Intrusions Means More Than BFT”.

Two workshops were co-located with the symposium and were held on 19 October 2012. The Workshop on Advances in Distributed Graph Algorithms (ADGA) was organized by Amos Korman. DISC’s Social Network Workshop (DISC’s SON) was organized by Anne-Marie Kermarrec and Alessandro Mei.

DISC 2012 acknowledges the use of the HotCRP system for handling submissions and managing the review process.

October 2012 Marcos K. Aguilera

Symposium Organization

DISC, the International Symposium on Distributed Computing, is an annual forum for the presentation of research on all aspects of distributed computing. It is organized in cooperation with the European Association for Theoretical Computer Science (EATCS). The symposium was established in 1985 as a biennial International Workshop on Distributed Algorithms on Graphs (WDAG). Its scope was soon extended to cover all aspects of distributed algorithms, and WDAG came to stand for International Workshop on Distributed Algorithms, becoming an annual symposium in 1989. In 1998, WDAG was renamed to DISC (International Symposium on Distributed Computing) to reflect the expansion of its coverage to all aspects of distributed computing, a field that has featured rapid and exciting developments.

Program Committee Chair

Marcos K. Aguilera	Microsoft Research Silicon Valley, USA

Program Committee

Lorenzo Alvisi	University of Texas at Austin, USA
James Aspnes	Yale University, USA
Hagit Attiya	Technion, Israel
Shlomi Dolev	Ben-Gurion University of the Negev, Israel
Faith Ellen	University of Toronto, Canada
Yuval Emek	ETH Zurich, Switzerland
Rui Fan	Nanyang Technological University, Singapore
Paola Flocchini	University of Ottawa, Canada
Felix Freiling	FAU, Germany
Cyril Gavoille	Université de Bordeaux, France
Seth Gilbert	National University of Singapore, Singapore
Fabíola Greve	Universidade Federal da Bahia, Brazil
Flavio Junqueira	Yahoo! Research, Spain
Petr Kuznetsov	TU Berlin/T Labs, Germany
Christoph Lenzen	Weizmann Institute, Israel
Toshimitsu Masuzawa	Osaka University, Japan
Boaz Patt-Shamir	Tel Aviv University, Israel
Andrzej Pelc	Université du Québec en Outaouais, Canada
Michel Raynal	IRISA, France
Eric Ruppert	York University, Canada
André Schiper	EPFL, Switzerland
Nir Shavit	MIT, USA and TAU, Israel

Neeraj Suri	TU Darmstadt, Germany
Philippas Tsigas	Chalmers University, Sweden
Jennifer Welch	Texas A&M University, USA
Shmuel Zaks	Technion, Israel
Piotr Zieliński	Google, USA

Steering Committee

Marcos K. Aguilera	Microsoft Research Silicon Valley, USA
Shlomi Dolev	Ben-Gurion University of the Negev, Israel
Antonio Fernández Anta	Institute IMDEA Networks, Spain
Chryssis Georgiou	University of Cyprus, Cyprus
Nancy Lynch	MIT, USA
David Peleg	Weizmann Institute, Israel
Sergio Rajsbaum (chair)	UNAM, Mexico

Local Organization

Raimundo Macêdo (Chair, Tutorial Chair)	Universidade Federal da Bahia, Brazil
Aline Andrade	Universidade Federal da Bahia, Brazil
Flávio Assis	Universidade Federal da Bahia, Brazil
Marcos Barreto	Universidade Federal da Bahia, Brazil
Sérgio Gorender	Universidade Federal da Bahia, Brazil

External Reviewers

Ittai Abraham
Beley Alexey
Dan Alistarh
Miguel Angel Mosteiro
Luciana Arantes
Sima Barak
Leonid Barenboim
Joffroy Beauquier
Hrishikesh B. Acharya
Martin Biely
Lelia Blin
Max Blin
Peter Bokor
François Bonnet
Zohir Bouzid
Armando Castañeda
Arnaud Casteigts
Keren Censor-Hillel
Bapi Chatterjee
Ioannis Chatzigiannakis
Wei Chen
Viacheslav Chernoy
Bogdan Chlebus
Gregory Chockler
Hyun Chul Chung
Allen Clement
Alejandro Cornejo
Shantanu Das
Carole Delporte-Gallet
Benjamin Doerr
Danny Dolev
Dana Drachsler
Lúcia Drummond
Ali Ebnenasir
Raphael Eidenbenz
Panagiota Fatourou
Hugues Fauconnier
Hillit Fisch
Mateo Frigo
Eli Gafni
Leszek Gasieniec
Georgios Georgiadis
Anders Gidenstam
Maria Potop-Butucaru
Vincent Gramoli
Rachid Guerraoui
Sandeep Hans
Danny Hendler
Maurice Herlihy
Ted Herman
Stephan Holzer
Damien Imbs
Taisuke Izumi
Tomoko Izumi

Marek Janicki
Colette Johnen
Tomasz Jurdzinski
Hirotsugu Kakugawa
Erez Kantor
Barbara Keller
Eliran Kenan
Amir Kimchi
Ralf Klasing
Guy Korland
Eleftherios Kosmas
Darek Kowalski
Evangelos Kranakis
Milind Kulkarni
Michael Kuperstein
Edya Ladan Mozes
Tobias Langner
Victor Luchangco
Matthias Majuntke
Alex Matveev
Alessia Milani
Avery Miller
Zarko Milosevic
Pradipta Mitra
Sébastien Monnet
Farnaz Moradi
Angelia Nedich
Dang Nhan Nguyen
Ioannis Nikolakopoulos
Fukuhito Ooshita
Rotem Oshman
Oren Othnay
Victor Pankratius
Ami Paz
David Peleg
Lucia Penso
Haim Peremuter
Franck Petit
Darko Petrovic
Laurence Pilard
Giuseppe Prencipe
Rami Puzis
Sergio Rajsbaum
Thomas Ropars
Gianluca Rossi
Jared Saia
Nuno Santos
Stav Sapir
Christian Scheideler
Elad Schiller
Stefan Schmid
Jochen Seidel
Marco Serafini
Hakan Sundell
Jukka Suomela
Shachar Timnat
Ruben Titos-Gil
Sébastien Tixeuil
Lewis Tseng
Nir Tzachar
Nitin Vaidya
David Wilson
Philipp Woelfel
Edmund Wong
Li Ximing
Amos Zamir
Akka Zemmari
Jin Zhang

Sponsoring Organizations

CAPES

European Association for Theoretical Computer Science

LaSiD at Universidade Federal da Bahia

Microsoft Research

Sociedade Brasileira de Computação

Table of Contents

CBTree: A Practical Concurrent Self-Adjusting Search Tree

Yehuda Afek[1], Haim Kaplan[1], Boris Korenfeld[1],
Adam Morrison[1], and Robert E. Tarjan[2]

[1] Blavatnik School of Computer Science, Tel Aviv University
[2] Princeton University and HP Labs

Abstract. We present the CBTree, a new *counting-based* self-adjusting binary search tree that, like *splay trees*, moves more frequently accessed nodes closer to the root. After m operations on n items, c of which access some item v, an operation on v traverses a path of length $\mathcal{O}(\log \frac{m}{c})$ while performing few if any rotations. In contrast to the traditional self-adjusting splay tree in which each accessed item is moved to the root through a sequence of tree rotations, the CBTree performs rotations infrequently (an amortized subconstant $o(1)$ per operation if $m \gg n$), mostly at the bottom of the tree. As a result, the CBTree scales with the amount of concurrency. We adapt the CBTree to a multicore setting and show experimentally that it improves performance compared to existing concurrent search trees on non-uniform access sequences derived from real workloads.

1 Introduction

The shift towards multicore processors raises the importance of optimizing concurrent data structures for workloads that arise *in practice*. Such workloads are often *non-uniform*, with some *popular* objects accessed more frequently than others; this has been consistently observed in measurement studies [1,2,3,4]. Therefore, in this paper we develop a concurrent data structure that completes operations on popular items faster than on ones accessed infrequently, leading to increased overall performance in practice.

We focus on the binary search tree (BST), a fundamental data structure for maintaining ordered sets. It supports successor and range queries in addition to the standard `insert`, `lookup` and `delete` operations.

To the best of our knowledge, no existing concurrent BST is *self-adjusting*, adapting its structure to the access pattern to provide faster accesses to popular items. Most concurrent algorithms (e.g., [5,6]) are based on sequential BSTs that restructure the tree to keep its height logarithmic in its size. The restructuring rules of these search trees do not prioritize popular items and therefore do not provide optimal performance for skewed and changing usage patterns.

Unlike these BSTs, sequential self-adjusting trees do not lend themselves to an efficient concurrent implementation. A natural candidate would be Sleator and Tarjan's seminal *splay tree* [7], which moves each accessed node to the root using a sequence of rotations called *splaying*. The amortized access time of a splay tree for an item v which is accessed $c(v)$ times in a sequence of m operations is $\mathcal{O}(\log \frac{m}{c(v)})$, asymptotically matching the information-theoretic optimum [8].

Unfortunately, splaying creates a major scalability problem in a concurrent setting. Every operation moves the accessed item to the root, turning the root into a hot spot, making the algorithm non-scalable. We discuss the limitations of some other sequential self-adjusting BSTs in Sect. 2. The bottom line is that no existing algorithm adjusts the tree to the access pattern *in practice* while still admitting a scalable concurrent implementation.

In this paper, we present a *counting-based tree*, CBTree for short, a self-adjusting BST that scales with the amount of concurrency, and has performance guarantees similar to the splay tree. The CBTree maintains a *weight* for each subtree S, equal to the total number of accesses to items in S. The CBTree operations use rotations in a way similar to splay trees, but rather than performing them at each node along the access path, decisions of where to rotate are based on the weights. The CBTree does rotations to guarantee that the weights along the access path decrease geometrically, thus yielding a path of logarithmic length. Specifically, after m operations, c of which access item v, an operation on v takes time $\mathcal{O}(1 + \log \frac{m}{c})$.

The CBTree's crucial performance property is that it performs only a *sub-constant* $o(1)$ amortized number of rotations per operation, so most CBTree operations perform few if any rotations. This allows the CBTree to scale with the amount of concurrency by avoiding the rotation-related synchronization bottlenecks that the splay tree experiences. Thus the performance gain by eliminating rotations using the counters outweighs losing the splay tree's feature of not storing book-keeping data in the nodes.

The CBTree replaces most of the rotations splaying does with counter updates, which are local and do not change the structure of the tree. To translate the CBTree's theoretical properties into good performance in practice, we minimize the synchronization costs associated with the counters. First, we maintain the counters using plain reads and writes, without locking, accepting an occasional lost update due to a race condition. We show experimentally that the CBTree is robust to inaccuracies due to data races on these counters.

Yet even plain counter updates are overhead compared to the read-only traversals of traditional balanced BSTs. Moreover, we observe that updates of concurrent counters can limit scalability on a multicore architecture where writes occur in a core's private cache (as in Intel's Xeon E7) instead of in a shared lower-level cache (as in Sun's UltraSPARC T2). We thus develop a *single adjuster* optimization in which an adjuster thread performs the self-adjusting as dictated by its own accesses and other threads do not update counters. When all the threads' accesses come from the same workload (same distribution), the adjuster's operations are representative of all threads, so the tree structure is good and performance improves, as most threads perform read-only traversals without causing serialization on counter cache lines. If the threads have different access patterns, the resulting structure will probably be poor no matter what, since different threads have different popular items.

In addition, we describe how to exponentially decay the CBTree's counters over time so that it responds faster to a change in the access pattern.

One can implement the CBTree easily on top of any concurrent code for doing rotations atomically, because the CBTree restructures itself by rotations as does any other BST. Our implementation uses Bronson et al.'s optimistic BST concurrency control technique [5]. We compare our CBTree implementation with other sequential and concurrent BSTs using real workloads, and show that the CBTree provides short access paths and excellent throughput.

2 Related Work

Treaps. A treap [9] is a BST satisfying the *heap property*: each node v also has a priority which is maximal in its subtree. An access to node v generates a random number r which replaces v's priority if it is greater than it, after which v is rotated up the treap until the heap property is restored. Treaps provide *probabilistic* bounds similar to those of the CBTree: node v is at expected depth $\mathcal{O}(\log \frac{m}{c(v)})$ and accessing it incurs an expected $\mathcal{O}(1)$ rotations [9]. Since the treap's rotations are driven by local decisions it is suitable for a concurrent implementation. However, we find that in practice nodes' depths in the treap vary from the expected bound. Consequentially, CBTrees (and splay trees) have better path length than treaps (Sect. 5).

Binary Search Trees of Bounded Balance. Nievergelt and Reingold [10] described how to keep a search tree balanced by doing rotations based on subtree sizes. Their goal was to maintain $O(\log n)$ height; they did not consider making more popular items faster to access.

Biased Search Trees. Several works in the sequential domain consider *biased* search trees where an item is *a priori* associated with a weight representing its access frequency. Bent, Sleator and Tarjan discuss these variants extensively [11]. Biased trees allow the item weight to be changed using a `reweight` operation, and can therefore be adapted to our dynamic setting by following each access with a `reweight` to increment the accessed item's weight. However, most biased tree algorithms do not easily admit an efficient implementation. In the biased trees of [11,12], for example, every operation is implemented using global tree splits and joins, and items are only stored in the leaves. The CBTree's rotations are the same as those in Baer's weight-balanced trees [13], but the CBTree uses different rules to decide when to apply rotations. Our analysis of CBTrees is based on the analysis of splaying whereas Baer did not provide a theoretical analysis. Baer also did not consider concurrency.

Concurrent Ordered Map Algorithms. Most existing concurrent BSTs are balanced and rely on locking, e.g. [5,6]. Ellen et al. proposed a nonblocking concurrent BST [14]. Their tree is not balanced, and their focus was obtaining a lock-free BST. Crain et al.'s recent transaction-friendly BST [15] is a balanced tree in which a dedicated thread continuously scans the tree looking for balancing rule violations that it then corrects. Unlike the CBTree's adjuster, this thread does not perform other BST operations. Ordered sets and maps can also be implemented using skip lists [16], which are also not self-adjusting.

3 The CBTree and Its Analysis

3.1 Splaying Analysis Background

The CBTree's design draws from the analysis of semi-splaying, a variant of splaying [7]. To (bottom-up) *semi-splay* an item v known to be in the tree, an operation first locates v in the tree in the usual way. It then ascends from v towards the root, restructuring the tree using rotations as it goes. At each *step*, the operation examines the next two nodes along the path to the root and decides which rotation(s) to perform according to the structure of the path, as depicted in Fig. 1. Following the rotation(s) it continues from the node which replaces the current node's grandparent in the tree (in Fig. 1, this is node y after a single rotation and node x after a double rotation). If only one node remains on the path to the root then the final edge on the path is rotated.

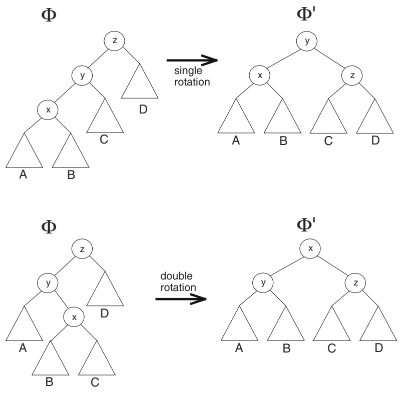


Fig. 1. Semi-splaying restructuring: current node is x, and the next two nodes on the path to the root are y and z. The case when y is a right child is symmetric.

To analyze the amortized performance of splaying and semi-splaying Sleator and Tarjan use the potential method [17]. The potential of a splay tree is defined based on an arbitrary but fixed positive weight which is assigned to each item.[1] The splay tree algorithm does not maintain these weights; they are defined for the analysis only.

Let $c(v)$ be the weight assigned to node v, and let $W(v)$ be the total weight of the nodes currently in the subtree rooted at v. Let $r(v) = \lg W(v)$ be the *rank* of v.[2] The *potential* of a splay tree is $\Phi = \sum r(v)$ over all nodes v in the tree. Sleator and Tarjan's analysis of semi-splaying relies on the following bound for the potential change caused by a rotation [7]:

Lemma 1. *Let Φ and Φ' be the potentials of a splay tree before and after a semi-splay step at node x, respectively. Let z be the grandparent of x before the semi-splay step (as in Fig. 1), and let $\Delta\Phi = \Phi' - \Phi$. Then*

$$2 + \Delta\Phi \leq 2(r(z) - r(x))$$

where $r(x)$ and $r(z)$ are the ranks of x and z in the tree before the step, respectively.

[1] The splay algorithm never changes the node containing an item, so we can also think of this as the weight of the node containing the item.

[2] We use lg to denote $\log_2$.

The analysis uses this lemma to show that the amortized time of semi-splaying node v in a tree rooted at *root* is $\mathcal{O}(r(root) - r(v)) = \mathcal{O}(\lg(W(root)/W(v)) + 1)$. This holds for any assignment of node weights; different selections of weights yield different bounds, such as bounds that depend on access frequencies of the items or bounds that capture other patterns in the access sequence. Here we focus on using a node's access frequency as its weight, i.e., given a sequence of m tree operations let $c(v)$ be the number of operations on v in the sequence. Using Lemma 1 with this weight assignment Sleator and Tarjan proved the following [7]:

Theorem 1 (Static Optimality of Semi-Splaying). *The total time for a sequence of m semi-splay operations on a tree with n items $v_1, \ldots, v_n$, where every item is accessed at least once, is*

$$\mathcal{O}\left(m + \sum_{i=1}^{n} c(v_i) \lg \frac{m}{c(v_i)}\right),$$

where $c(v)$ is the number of times v is accessed in the sequence.

Hereafter we say that $\mathcal{O}(\lg(m/c(v)) + 1)$ is the *ideal access time* of v.

3.2 The Sequential CBTree

A CBTree is a binary search tree where each node contains an item; we use the terms *item* and *node* interchangeably. We maintain in any node v a *weight* $W(v)$ which counts the total number of operations on v and its descendants. We can compute $C(v)$, the number of operations performed on v until now, from the weight of v and its children by $C(v) = W(v) - (W(v.left) + W(v.right))$, using a weight of 0 for null children.

A CBTree operation performs steps similar to semi-splaying, however it does them in a top-down manner and it decides whether to perform rotations based on the weights of the relevant nodes.

Lookup: To lookup an item v we descend from the root to v, possibly making rotations on the way down. We maintain a current node z along the path to v and look two nodes ahead along the path to decide whether to perform a single or a double rotation. Assume that the next node y along the path is the left child of z (the case when it is a right child is symmetric). If the child of y along the path is also a left child we may decide to perform a single rotation as in the top part of Fig. 1. If the child of y along the path is a right child we may perform a double rotation as in the bottom part of Fig. 1. From here on, unless we need to distinguish between the cases, we refer to a single or a double rotation simply as a *rotation.*

We perform a rotation only if it would decrease the potential of the tree by at least a positive constant $\epsilon \in (0, 2)$, i.e., if $\Delta\Phi < -\epsilon$. After performing a rotation at z, the CBTree operation changes the current node to be the node that replaces z in the tree, i.e., node y after a single rotation or node x after a double rotation. If we do not perform a rotation, the current node *skips ahead* from z to x without restructuring the tree. (Naturally, if the search cannot continue past y the search moves to y instead.) A search whose traversal ends finding the desired item v increments $W(v)$ and then proceeds in a bottom-up manner to increment the weights of all the nodes on the path from v to the root.

To summarize, during a search the parent of the current node stays the same and we keep reducing the potential by at least ϵ using rotations until it is no longer possible. We then advance the current node two steps ahead to its grandchild.

Insert: An `insert` of v searches for v while restructuring the tree, as in a lookup. If v is not found, we replace the null pointer where the search terminates with a pointer to a new node containing v with $W(v) = 1$, then increment the weights along the path from v to the root. If v is found we increment $W(v)$ and the weights along the path and consider the `insert` failed.[3]

Delete: We `delete` an item v by first searching for it while restructuring the tree as in a `lookup`. If v is a leaf we unlink it from the tree. If v has a single child we remove v from the tree by linking v's parent to v's child. If v has two children we only mark it as deleted. We adapt all operations so that any restructuring which makes a deleted node a leaf or a single child unlinks it from the tree. With these changes, CBTree's space consumption remains linear in n, the number of non-deleted nodes in the tree. If we are willing to count failed `lookup`s as accesses, we can update the counters top-down instead of bottom-up.

Computing Potential Differences: To decide whether to perform a rotation, an operation needs to compute $\Delta\Phi$, the potential difference resulting from the rotation. It can do this using only the counters of the nodes involved in the rotation, as follows. Consider first the single rotation case of Fig. 1 (the other cases are symmetric). Only nodes whose subtrees change contribute to the potential change, so $\Delta\Phi = r'(z) + r'(y) - r(z) - r(y)$, where $r'(v)$ denotes the rank of v after the rotation. Because the overall weight of the subtree rooted at z does not change, $r'(y) = r(z)$, and thereby $\Delta\Phi = r'(z) - r(y)$. We can express $r'(z)$ using the nodes and their weights in the tree before the rotation to obtain

$$\Delta\Phi = \lg(C(z) + W(y.right) + W(z.right)) - \lg W(y). \tag{1}$$

For a double rotation, an analogous derivation yields

$$\begin{aligned}\Delta\Phi = &\lg\left(C(z) + W(x.right) + W(z.right)\right) + \\ &\lg\left(C(y) + W(y.left) + W(x.left)\right) - \\ &\lg W(y) - \lg W(x).\end{aligned} \tag{2}$$

When we do a rotation, we also update the weights of the nodes involved in the rotation.

Note that our implementation works with the weights directly by computing $2^{\Delta\Phi}$ using logarithmic identities and comparing it to $2^{-\epsilon}$.

[3] A failed `insert()` can change auxiliary information associated with v.

An alternative rule for deciding when to do a rotation is to rotate when $W(z)/W(x) < \alpha$, where α is a constant less than two. This rule is simpler to apply, simpler to analyze, and more intuitive than rotating when the potential drops by at least ϵ, but we have not yet had time to try it in experiments.

3.3 Analysis

In this section we consider only *successful* lookups, that is, lookups of items that are in the tree. For simplicity, we do not consider deleting an item v and then later inserting it again, although the results can be extended by considering such an item to be a new item. We prove that Theorem 1 holds for the CBTree.

Theorem 2. *Consider a sequence of m operations on n distinct items, $v_1, \ldots,$ v_n, starting with an initially empty CBTree. Then the total time it takes to perform the sequence is*

$$\mathcal{O}\left(m + \sum_{i=1}^{n} c(v_i) \lg \frac{m}{c(v_i)}\right),$$

where $c(v)$ is the number of times v is accessed in the sequence.

An operation on item v does two kinds of steps: (1) rotations, and (2) traversals of edges in between rotations. The edges traversed in between rotations are exactly the ones on the path to v at the end of the operation. Our proof of Theorem 2 accounts separately for the total time spent traversing edges in between rotations and the total time spent doing rotations.

We first prove that an operation on node v, applied to a CBTree with weights $W(u)$ for each node u, traverses $\mathcal{O}(\lg(W/C(v))$ edges in between rotations, where $W = W(root)$ is the weight of the entire tree and $C(v)$ is the individual weight of v at the time the operation is performed (Lemma 2). From this we obtain that the time spent traversing edges between rotations is $\mathcal{O}(c(v) + c(v) \lg \frac{m}{c(v)})$ (Lemma 3). Having established this, the amortized bound in Theorem 2 follows by showing that the total number of rotations in all m operations in the sequence is $\mathcal{O}(n + n \ln \frac{m}{n}) = \mathcal{O}(m)$.

Lemma 2 (Ideal access path). *Consider a CBTree with weights $W(u)$ for each node u. The length of the path traversed by an operation on item v is $\mathcal{O}(\lg(W/C(v)) + 1)$, where $W = W(root)$ and $C(v)$ is the individual weight of v, at the time the operation is performed.*

Proof. The path to v at the end of the operation (i.e., just before v is unlinked if the operation is a `delete`) consists of d pairs of edges (z, y) and (y, x) that the operation skipped between rotations, and possibly a single final edge if the operation could look ahead only one node at its final step. For each such pair (z, y) and (y, x), let $\Delta\Phi$ be the potential decrease we would have got by performing a rotation at z. Since we have not performed a rotation, $\Delta\Phi > -\epsilon$. By Lemma 1 we obtain that

$$2(r(z) - r(x)) \geq 2 + \Delta\Phi > 2 - \epsilon. \tag{3}$$

Define $\delta = 1 - \epsilon/2$. Then Equation (3) says that $r(z) - r(x) > \delta$ for each such pair of edges (z, y) and (y, x) on the final path. Summing over the d pairs on the path we get that $r(root) - r(v) > d\delta$ and so

$$d < \frac{r(root) - r(v)}{\delta} = \mathcal{O}\left(\lg \frac{W(root)}{W(v)}\right) = \mathcal{O}\left(\lg \frac{W}{C(v)}\right).$$

Since the length of path to v is at most $2d + 1$, the lemma follows. □

We now show that Lemma 2's bound matches that of the splay tree.

Lemma 3. *The total time spent traversing edges between rotations in all the operations that access v is $\mathcal{O}(c(v) + c(v) \lg \frac{m}{c(v)})$.*

Proof. The time spent traversing edges between rotations is 1 plus the number of edges traversed. Because the CBTree's weight is at most m throughout the sequence, and v's individual weight after the k-th time it is accessed is k, Lemma 2 implies that the time spent traversing edges between rotations over all the operations accessing v is $c(v) + \mathcal{O}\left(\sum_{j=1}^{c(v)} \lg \frac{m}{j}\right)$. By considering separately the cost of the final half of the operations, the quarter before it, and so on, we bound this as follows:

$$\begin{aligned} & c(v) + \mathcal{O}\left(\frac{c(v)}{2} \lg \frac{m}{c(v)/2} + \frac{c(v)}{4} \lg \frac{m}{c(v)/4} + \dots\right) \\ =& c(v) + \mathcal{O}\left(c(v) \lg \frac{m}{c(v)} + c(v)(\frac{\lg 2}{2} + \frac{\lg 4}{4} + \dots)\right), \end{aligned}$$

which is $\mathcal{O}\left(c(v) + c(v) \lg \frac{m}{c(v)}\right)$ because $\sum_{k=1}^{\infty} \frac{k}{2^k} = 2$. □

The following lemma, proved in the extended version of the paper [18], bounds the number of rotations.

Lemma 4. *In a sequence of m operations starting with an empty CBTree, we perform $O(n + n \ln \frac{m}{n}) = \mathcal{O}(m)$ rotations, where n is the number of insertions.*

4 The Concurrent CBTree

We demonstrate the CBTree using Bronson et al.'s optimistic BST concurrency control technique [5] to handle synchronization of generic BST operations, such as rotations and node link/unlinks. To be self contained, we summarize this technique in Sect. 4.1. Section 4.2 then describes how we incorporate the CBTree into it; due to space limitations, pseudo-code is presented in the extended version of the paper [18]. Section 4.3 describes our *single-adjuster* optimization. Section 4.4 describes the Lazy CBTree, a variant of the CBTree meant to reduce the overhead caused by calculating potential differences during the traversal.

4.1 Optimistic Concurrent BSTs

Bronson et al. implement traversal through the tree without relying on read-write locks, using *hand-over-hand optimistic validation*. This is similar to hand-over-hand locking [19] except that instead of overlapping lock-protected sections, we overlap atomic blocks which traverse node links. Atomicity within a block is established using versioning. Each node holds a version number with reserved `changing` bits to indicate that the node is being modified. A navigating reader (1) reads the version number and waits for the `changing` bits to clear, (2) reads the desired fields from the node, (3) rereads the version. If the version has not changed, the reads are atomic.

4.2 Concurrent CBTree Walk-Through

We represent a node v's weight $W(v)$, equal to the total number of accesses to v and its descendants, with three counters: $selfCnt$, counting the total number of accesses to v, $rightCnt$ for the total number of accesses to items in v's right subtree, and $leftCnt$, an analogous counter for the left subtree.

Traversal: Hand-over-hand validation works by chaining short atomic sections using recursive calls. Each section traverses through a single node u after validating that both the *inbound* link, from u's parent to u, and the *outbound* link, from u to the next node on the path, were valid together at the same point in time. If the validation fails, the recursive call returns so that the previous node can revalidate itself before trying again. Eventually the recursion unfolds bottom-up back to a consistent state from which the traversal continues.

When traversing through a node (i.e., at each recursive call) the traversal may perform a rotation. We describe the implementation of rotations and of the rotation decision rule below. For now, notice that performing a rotation invalidates both the inbound and outbound links of the current node. Therefore, after performing a rotation the traversal returns to the previous recursion step so that the caller revalidates itself. Using Fig. 1's notation, after performing a rotation at z the recursion returns to z's parent (previous block in the recursion chain) and therefore continues from the node that replaces z in the tree. In doing this, we establish that a traversed link is always verified by the hand-over-hand validation, as in the original optimistic validation protocol. The linearizability [20] of the CBTree therefore follows from the linearizability of Bronson et al.'s algorithm.

Rotations: To do a rotation, the CBTree first acquires locks on the nodes whose links are about to change in parent-to-child order: z's parent, z, y, and for a double rotation also x (using Fig. 1's notation). It then validates that the relationship between the nodes did not change and performs the rotation which is done exactly as in Bronson et al.'s algorithm, changing node version numbers as required and so on. After the rotation the appropriate counters are updated to reflect the number of accesses to the new subtrees.

Counter Maintenance: Maintaining consistent counters requires synchronizing with concurrent traversals and rotations. While traversals can synchronize by atomically incrementing the counters using `compare-and-swap`, this does not solve the coordination problem between rotations and traversals, and any additional mechanism to synchronize them would be pure overhead because rotations are rare. We therefore choose to access counters using plain reads and writes, observing that wrong counter values will not violate the algorithm's correctness, only possibly its performance.

A traversal increments the appropriate counters as the recursion unfolds at the end of the operation (i.e., not during the intermediate retries that may occur). In the extended version [18] we discuss the races that can occur with this approach and show that such races – if they occur – do not keep the CBTree from obtaining short access paths for frequent items.

Operation Implementation: A `lookup` is a traversal. Insertion is a traversal that terminates by adding a new item or updating the current item's value. Our `delete` implementation follows Bronson et al.'s approach [5], marking a node as deleted and unlinking it when it has one or no children.

Speeding Up Adaptation to Access Pattern Change: After a change in the access pattern, i.e., when a popular item becomes unpopular, frequent nodes from the new distribution may take a lot of time until their counters are high enough to beat nodes that lost their popularity. To avoid this problem we add an exponential decay function to the counters, based on an *external clock* that is updated by an auxiliary thread or by the hardware. We detail this technique in the extended version [18]. We note here that the decaying is an infrequent event performed by threads as they traverse the tree. Therefore decaying updates can also be lost due to races, which we again accept since the decaying is an optimization that has no impact on correctness.

4.3 Single Adjuster

Even relaxed counter maintenance can still degrade scalability on multicore architectures such as Intel's Xeon E7, where a memory update occurs in a core's private cache, after the core acquires exclusive ownership of the cache line. In this case, when all cores frequently update the same counters (as happens at the top of the tree) each core invalidates a counter's cache line from its previous owner, who in turn had to take it from another core, and so on. On average, a core waits for all other cores to acquire the cache line before its write can complete. In contrast, on Sun's UltraSPARC T2 Plus architecture all writes occur in a shared L2 cache, allowing the cores to proceed quickly: the L2 cache invalidates all L1 caches in parallel and completes the write.

To bypass this Intel architectural limitation, we propose an optimization in which only a single *adjuster* thread performs counter updates during its `lookup`s. The remaining threads perform read-only `lookup`s. Thus, only the adjuster thread requires exclusive ownership of counter cache lines; other `lookup`s request shared ownership, allowing their cache misses to be handled in parallel. Similarly, when the adjuster writes to a counter, the hardware sends the invalidation requests in parallel. Synchronization can be further reduced by periodically switching the adjuster to read-only mode, as we did in our evaluation.

4.4 The Lazy CBTree

Calculating potential differences during the CBTree traversal, which involves multiplication and division instructions, has significant cost on an in-order processor such as Sun's UltraSPARC T2. When no rotation is performed – as is usually the case – the calculations are pure overhead. We have therefore developed the Lazy CBTree, a variant of the CBTree that greatly reduces this overhead by not making rotation decisions during a `lookup` traversal.

A Lazy CBTree traversal makes no rotation decisions along the way. When reaching the destination node, the operation makes a single rotation decision which is based only on counter comparisons, and then proceeds to update the counters along the path back to the root. If the operation is an `insert()` of a new item, it may make more (cheap) rotation decisions as it unfolds the path back to the root. We refer the reader to the extended version for details [18]. While the CBTree analysis does not apply to the Lazy CBTree, in practice the Lazy CBTree obtains comparable path lengths to CBTree but with much lower cost per node traversal, and so obtains better overall throughput.

5 Experimental Evaluation

In this section we compare the CBTree's performance to that of the splay tree, treap [9] and AVL algorithms. All implementations are based on Bronson et al.'s published source code. Benchmarks are run on a Sun UltraSPARC T2+ processor and on an Intel Xeon E7-4870 processor. The UltraSPARC T2+ (Niagara II) is a multithreading (CMT) processor, with 8 1.165 HZ in-order cores with 8 hardware strands per core, for a total of 64 hardware strands per chip. Each core has a private L1 write-through cache and the L2 cache is shared. The Intel Xeon E7-4870 (Westmere EX) processor has 10 2.40GHz cores, each multiplexing 2 hardware threads. Each core has private write-back L1 and L2 caches and a shared L3 cache.

Overall, we consider the following implementations: (1) **CB**, CBTree with decaying of node counters disabled, (2) **LCB**, the lazy CBTree variant (Sect. 4.4), (3) **Splay**, Daniel Sleator's sequential top-down splay tree implementation [21] with a single lock to serialize all operations, (4) **Treap**, and (5) **AVL**, Bronson et al.'s relaxed balance AVL tree [5]. Because our single adjuster technique applies to any self-adjusting BST, we include single adjuster versions of the splay tree and treap in our evaluation, which we refer to as **[Alg]OneAdjuster** for **Alg** $\in$ {Splay,Treap,CB,LCB}. In these implementations one dedicated thread alternates between doing lookups as in **Alg** for 1 millisecond and lookups without restructuring for t milliseconds ($t = 1$ on the UltraSPARC and $t = 10$ on the Intel; these values produced the best results overall). All other threads always run lookups without any restructuring. Insertions and deletions are done as in **Alg** for all threads. Note that SplayOneAdjuster is implemented using Bronson et al.'s code to allow `lookup`s to run concurrently with the adjuster's rotations.

5.1 Realistic Workloads

Here the algorithms are tested on access patterns derived from real workloads: (1) **books**, a sequence of 1,800,825 words (with 31,779 unique words) generated by concatenating ten books from Project Gutenberg [22], (2) **isp**, a sequence of 27,318,568 IP addresses (449,707 unique) from packets captured on a 10 gigabit/second backbone link of a Tier1 ISP between Chicago, IL and Seattle, WA in March, 2011 [23], and (3) **youtube**, a sequence of 1,467,700 IP addresses (39,852 unique) from YouTube user request data collected in a campus network measurement [24]. As the traces we obtained are of item sequences without accompanying operations, in this test we use only `lookup` operations on the items in the trace, with no `inserts` or `deletes`. To avoid initialization effects each algorithm starts with a maximum balanced tree over the domain, i.e., where the median item is the root, the first quartile is the root's left child, and so on. Each thread then repeatedly acquires a 1000-operation chunk of the sequence and invokes the operations in that subsequence in order, wrapping back to the beginning of the sequence after the going through entire sequence.

Table 1 shows the average number of nodes traversed and rotations done by each operation on the Sun UltraSPARC machine. As these are algorithmic rather than implementation metrics, results on the Intel are similar and thus omitted. Figure 2 shows the throughput, the number of operations completed by all the threads during the duration of the test.

CBTree obtains the best path length, but this does not translate to the best performance: on the UltraSPARC, while CBTree scales well, its throughput is

Table 1. Average path length and number of rotations for a single thread and 64 threads. When the single thread result, r_1, significantly differs from the 64 threads result, r_{64}, we report both as r_1, r_{64}. To reduce overhead, data is collected by one representative thread, who is the adjuster in the single adjuster variants.

	AVL	Splay	Treap	CBTree	Lazy CBTree	Splay	Treap	CBTree	Lazy CBTree
						Single adjuster			
Average path length									
books	17.64	10.63, 12.06	11.19	**9.54**, **8.64**	**9.71**, **9.11**	11.71 11.71	11.43 11.43	**10.13**, **11.06**	**10.30**, **10.68**
isp	17.86	9.09, 12.89	14.64	**11.13**	**11.95**, **11.38**	13.39	14.46, 15.1	**12.33**, **13.25**	**12.41**, **13.49**
youtube	14.35	8.52, 13.31	15.75	**11.81**	**12.06**, **11.87**	13.47	15.84	**12.27**	**12.24**
Rotations per operation									
books	0	9.16	< 0.01	$<$ **0.01**	**0.02**	2.95	0.09	**0.02**	**0.02**
isp	0	8.09, 10.45	0.03	**0.01**	**0.03**	2.65, 3.10	0.25	**0.01**	**0.03**
youtube	0	7.52, 10.73	< 0.01	$<$ **0.01**	$<$ **0.01**	3.17	0.11	**0.03**	**0.02**

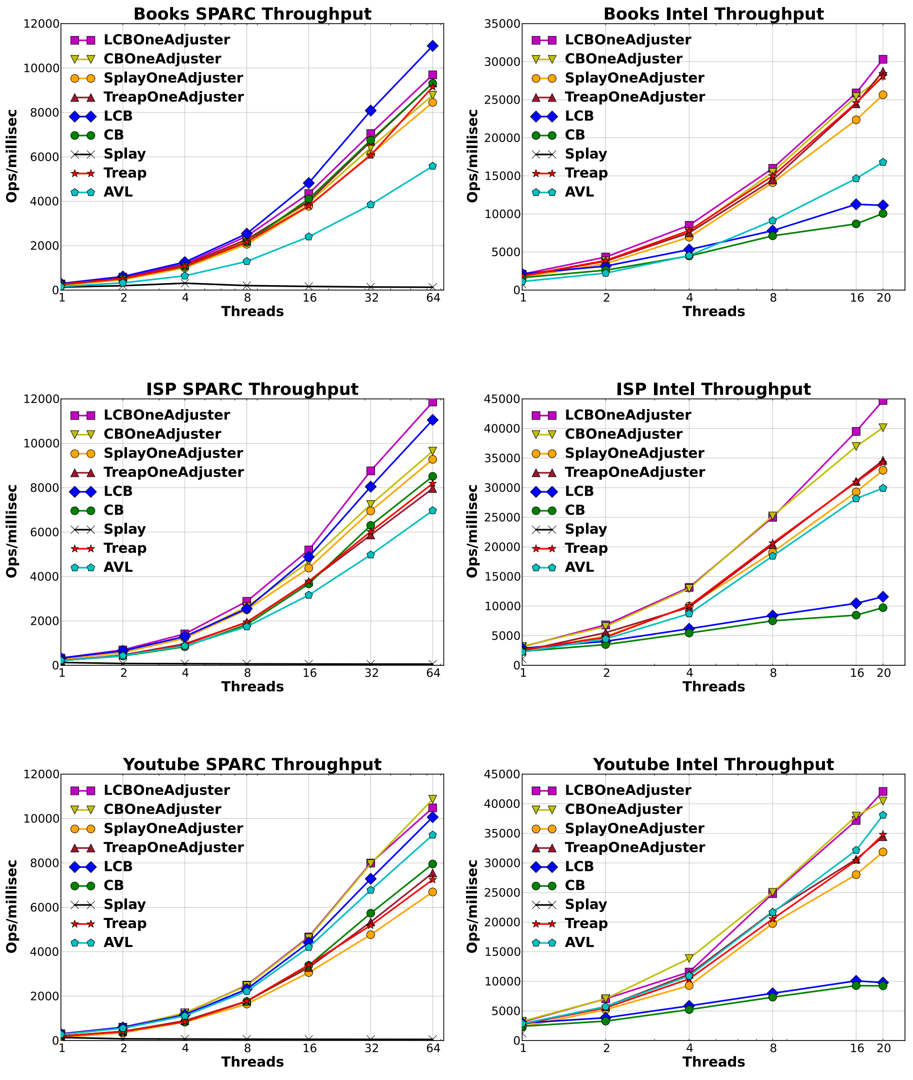


Fig. 2. Test sequence results. **Left:** Sun UltraSPARC T2+ (up to 64 hardware threads). **Right:** Intel Westmere EX (up to 20 hardware threads).

lower than some of the other algorithms due to the cost of calculating potential differences, and on the Intel Xeon E7 CBTree scales poorly because the counter updates serialize the threads (Sect. 4.3). Lazy CBTree, which avoids computing potential differences, outperforms all algorithms except single adjuster variants on the UltraSPARC, but also scales poorly on the Intel.

The single adjuster solves the above problems, making CBOneAdjuster and LCBOneAdjuster the best performers on both architectures. For example, on the **isp** sequence, CBOneAdjuster and LCBOneAdjuster outperform treap, the next best algorithm, respectively by 15% and 30% on the Intel and by 20% and 50% on the UltraSPARC at maximum concurrency. Because Lazy CBTree on the UltraSPARC incurs little overhead, if LCBOneAdjuster obtains significantly worse path length (e.g., on the **books** sequence), it performs worse than Lazy CBTree. The treap high performance is because an operation usually updates only its target node. However, this results in suboptimal path lengths, and also prevents the treap from seeing much benefit due to the single adjuster technique. While the AVL tree scales well, its lack of self-adjusting leads to suboptimal path lengths and performance. On **books**, for example, CBOneAdjuster outperforms AVL by 1.6× at 64 threads on the UltraSPARC machine.

The splay's tree coarse lock prevents it from translating its short path length into actual performance. Applying our single adjuster optimization allows readers to run concurrently and benefit from the adjuster's splaying, yielding a scalable algorithm with significantly higher throughput. Despite obtaining comparable path lengths to CBOneAdjuster, the SplayOneAdjuster does $> 100\times$ more rotations than CBOneAdjuster, which force the concurrent traversals to retry. As a result, CBOneAdjuster outperforms SplayOneAdjuster.

Additional Experiments: The extended version [18] describes additional experiments: (1) evaluating the algorithms on synthetic skewed workloads following a Zipf distribution, (2) examining how the algorithms adjust to changes in the usage pattern, (3) measuring performance under different ratios of `insert`/`delete`/`lookup`, and (4) showing that CBTree is robust to lost counter updates.

Future work. We intend to experiment with other CBTree variants, including the one mentioned at the end of Sect. 3.2, as well as with a top-down version of semi-splaying.

Acknowledgments. This work was supported by the Israel Science Foundation under grant 1386/11, by the Israeli Centers of Research Excellence (I-CORE) program (Center 4/11), and by the BSF. Robert E. Tarjan's work at Princeton is partially supported by NSF grant CCF-0832797. Adam Morrison is supported by an IBM PhD Fellowship.

References

1. Gill, P., Arlitt, M., Li, Z., Mahanti, A.: YouTube traffic characterization: a view from the edge. In: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement, IMC 2007, pp. 15–28. ACM, New York (2007)
2. Mahanti, A., Williamson, C., Eager, D.: Traffic analysis of a web proxy caching hierarchy. IEEE Network 14(3), 16–23 (2000)
3. Cherkasova, L., Gupta, M.: Analysis of enterprise media server workloads: access patterns, locality, content evolution, and rates of change. IEEE/ACM Transactions on Networking 12(5), 781–794 (2004)
4. Sripanidkulchai, K., Maggs, B., Zhang, H.: An analysis of live streaming workloads on the internet. In: Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, IMC 2004, pp. 41–54. ACM, New York (2004)

5. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010, pp. 257–268. ACM, New York (2010)
6. Hanke, S., Ottmann, T., Soisalon-soininen, E.: Relaxed Balanced Red-black Trees. In: Bongiovanni, G., Bovet, D.P., Di Battista, G. (eds.) CIAC 1997. LNCS, vol. 1203, pp. 193–204. Springer, Heidelberg (1997)
7. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. Journal of the ACM 32, 652–686 (1985)
8. Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City
9. Seidel, R., Aragon, C.R.: Randomized search trees. Algorithmica 16, 464–497 (1996), doi:10.1007/s004539900061
10. Nievergelt, J., Reingold, E.M.: Binary search trees of bounded balance. In: Proceedings of the Fourth Annual ACM Symposium on Theory of Computing, STOC 1972, pp. 137–142. ACM, New York (1972)
11. Bent, S.W., Sleator, D.D., Tarjan, R.E.: Biased 2-3 trees. In: Proceedings of the 21st Annual Symposium on Foundations of Computer Science, FOCS 1980, pp. 248–254. IEEE Computer Society, Washington, DC (1980)
12. Feigenbaum, J., Tarjan, R.E.: Two new kinds of biased search trees. Bell System Technical Journal 62, 3139–3158 (1983)
13. Baer, J.L.: Weight-balanced trees. In: American Federation of Information Processing Societies: 1975 National Computer Conference, AFIPS 1975, pp. 467–472. ACM, New York (1975)
14. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: Proceeding of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC 2010, pp. 131–140. ACM, New York (2010)
15. Crain, T., Gramoli, V., Raynal, M.: A speculation-friendly binary search tree. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2012, pp. 161–170. ACM, New York (2012)
16. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Communications of the ACM 33, 668–676 (1990)
17. Tarjan, R.E.: Amortized computational complexity. SIAM Journal on Algebraic and Discrete Methods 6(2), 306–318 (1985)
18. Afek, Y., Kaplan, H., Korenfeld, B., Morrison, A., Tarjan, R.E.: CBTree: A practical concurrent self-adjusting search tree. Technical report (2012)
19. Bayer, R., Schkolnick, M.: Concurrency of operations on b-trees. In: Readings in Database Systems, pp. 129–139. Morgan Kaufmann Publishers Inc., San Francisco (1988)
20. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS) 12, 463–492 (1990)
21. Sleator, D.D.: Splay tree implementation, http://www.link.cs.cmu.edu/splay
22. Project Gutenberg, http://www.gutenberg.org/
23. kc claffy, Andersen, D., Hick, P.: The CAIDA anonymized 2011 internet traces, http://www.caida.org/data/passive/passive_2011_dataset.xml
24. Zink, M., Suh, K., Gu, Y., Kurose, J.: Watch global, cache local: YouTube network traffic at a campus network - measurements and implications. In: Proceeding of the 15th SPIE/ACM Multimedia Computing and Networking Conference, vol. 6818 (2008)

Efficient Fetch-and-Increment[⋆]

Faith Ellen[1], Vijaya Ramachandran[2], and Philipp Woelfel[3]

[1] University of Toronto
faith@cs.toronto.edu
[2] University of Texas at Austin
vlr@cs.utexas.edu
[3] University of Calgary
woelfel@ucalgary.ca

Abstract. A Fetch&Inc object stores a non-negative integer and supports a single operation, fi, that returns the value of the object and increments it. Such objects are used in many asynchronous shared memory algorithms, such as renaming, mutual exclusion, and barrier synchronization. We present an efficient implementation of a wait-free Fetch&Inc object from registers and load-linked/store-conditional (ll/sc) objects. In a system with p processes, every fi operation finishes in $O(\log^2 p)$ steps, and only a polynomial number of registers and $O(\log p)$-bit ll/sc objects are needed. The maximum number of fi operations that can be supported is limited only by the maximum integer that can be stored in a shared register. This is the first wait-free implementation of a Fetch&Inc object that achieves both poly-logarithmic step complexity and polynomial space complexity, but does not require unrealistically large ll/sc objects or registers.

1 Introduction

A Fetch&Inc object stores a non-negative integer and supports a single operation, fi, that returns the value of the object and increments it. Such objects are fundamental synchronization primitives which have applications in many asynchronous shared memory algorithms. For example, a one-shot Fetch&Inc object, which allows at most one fi operation per process, can be used to solve the one-shot renaming problem: assign unique names from a small name space to participating processes. Each participating process performs fi and uses the result as its name. Thus, if k processes participate, they get unique names in the optimal range $\{0, \ldots, k-1\}$. Fetch&Inc objects have also been used in algorithms for mutual exclusion [5], barrier synchronization [10], work queues [11], and producer/consumer buffers [12,6].

We consider *wait-free*, *linearizable* implementations of Fetch&Inc objects in the standard asynchronous shared memory system with p processes with unique

[⋆] This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by Grant CCF-0830737 of the US National Science Foundation (NSF).

identifiers, $1, \ldots, p$. Wait-freedom means that each FETCH&INC operation finishes within a finite number of (its own) steps. Linearizability imposes the condition that when some instance *op* of FI returns the value v, the total number of completed FI operations (including *op*) is at most $v + 1$, and the total number of completed and pending FI operations is at least $v + 1$.

FETCH&INC objects have consensus number two, which means that they can be used to solve wait-free consensus for two processes, but not three. It is not possible to implement FETCH&INC objects just from registers. This is in contrast to *weak counter* objects, which support two separate operations, INCREMENT and READ, where INCREMENT increases the value of the counter by one but does not return anything, and READ returns the counter value. Unlike FETCH&INC objects, weak counters have wait-free implementations from registers. Our FETCH&INC implementation also supports a READ operation that returns the object value and, thus, is strictly stronger than a weak counter.

To implement FETCH&INC objects, the system needs to provide primitives of consensus number at least two. Implementations from TEST&SET and SWAP objects exist [2], but are inefficient. In fact, a lower bound by Jayanti, Tan, and Toueg [17] implies that for any weak counter implementation from resettable consensus and arbitrary history-less objects (and thus from TEST&SET and SWAP objects), some operations may require $\Omega(p)$ shared memory accesses. However, non-linearizable counters, such as those obtained from counting networks [6], can be more efficient. But a linear lower bound on the depth of linearizable counting networks [15] shows that such networks cannot be used to obtain efficient linearizable FETCH&INC implementations.

We consider implementations of FETCH&INC from load-linked/store-conditional (LL/SC) objects. An LL/SC object O provides three operations: LL, VL, and SC. LL(O) returns the value of object O. VL(O) returns TRUE or FALSE and, like LL(O), does not change the value of the object. SC(O, x) either sets the value of object O to x and returns TRUE or does not change the value of O and returns FALSE. A VL(O) or SC(O, x) operation by process p returns TRUE (in which case, we say that it is successful) if and only if p previously executed LL(O) and no other process has executed a successful SC on object O since p's last LL(O).

LL/SC objects allow implementations of any properly specified object using universal constructions. However, such generic universal constructions are not efficient. For example, Herlihy's standard universal constructions [13,14] require $\Omega(p)$ steps per implemented operation. As pointed out by Jayanti [16], the universal construction by Afek, Dauber and Touitou [1] can be modified so that each implemented operation takes only $O(\log p)$ steps, which is optimal. But this requires that registers can hold enough information to describe p operations. Since the description of an operation includes the identifier of the process that is executing the operation, $\Omega(p \log p)$-bit registers are necessary. Thus, this construction is impractical for systems with many processes. There are also efficient randomized FETCH&INC implementations (e.g., Alistarh etal. presented one based on repeated randomized renaming [3]), but there seems to be no obvious way to derandomize them.

In this paper, we present two FETCH&INC implementations that have polylogarithmic (in p) step complexity and do not require unrealistically large registers or LL/SC objects. In particular, $O(\log p)$ bits suffice for each LL/SC object and registers just need to be large enough to store the value of the FETCH&INC object. Our first implementation, presented in Section 2, is efficient when the number of FI operations, n, is polynomial in the number of processes. Each FI operation finishes in $O\big((\log p)(\log n)\big)$ steps, and a total of $O\big(p+n(\log p)(\log n)\big)$ shared registers and LL/SC objects are used. Then, in Section 3, we will explain how to extend this implementation (using a memory compression technique) to improve the worst case step complexity to $O\big((\log p)^2\big)$, using $O(p^3)$ shared registers and LL/SC objects. Both of our implementations support a READ operation with constant step complexity.

2 The First Implementation

The idea of our first implementation is that processes cooperate to construct (an implicit representation of) a sequence of process identifiers. The sequence has one copy of i for each instance of FI that process i performs. The values returned by these instances are the positions of i within this sequence, in increasing order.

The main data structure is a fixed balanced binary tree τ with p leaves, one per process, and height $\lceil \log p \rceil$. The representation of τ doesn't matter. For example, it can be stored implicitly in an array, like a binary heap. Let $P(v)$ denote the set of ids of processes whose leaves are in the subtree rooted at node v. At each node, v, there is an implicit representation of a sequence, $C(v)$, of ids in $P(v)$. Initially, $C(v)$ is empty. The sequence $C(v)$ at an internal node is an interleaving of a prefix of the sequence $C(left(v))$ at its left child and a prefix of the sequence $C(right(v))$ at its right child.

To perform FI, process i appends i to the sequence at process i's leaf. Then process i proceeds up the tree, trying to propagate information about the sequence at the current node, v, and the sequence at its sibling to its parent, as in [1]. It combines the current information at v and $sibling(v)$ and then tries to change $parent(v)$ so that it contains this updated information. If it doesn't succeed, it tries again. If it doesn't succeed a second time, it is guaranteed that some other process has already propagated the necessary information to $parent(v)$. Process i determines the position of its instance in $C(parent(v))$, the sequence at the parent of its current node, from the position of its instance in $C(v)$ and the number of elements from its $sibling(v)$ that precede the block containing its instance. Then process i moves to $parent(v)$. When process i reaches the root, it returns the position of its instance in the sequence at the root. The sequence $C(root(\tau))$ provides a linearization of all completed instances of FI and at most one uncompleted instance of FI by each process.

The sequence at process i's leaf is represented by a single-writer register, N_i, containing the length of the sequence. Thus $N_i = 0$ if the sequence at this leaf is empty. To append i to the sequence at this leaf, process i simply increments the value of N_i.

For any internal node, v, let $N(v)$ denote the length of the sequence $C(v)$ at v, let $NL(v)$ denote the number of elements of $C(v)$ whose leaves are in v's left subtree, and $NR(v)$ denote the number of elements of $C(v)$ whose leaves are in v's right subtree. Then $N(v) = NL(v) + NR(v)$. The sequence $C(v)$ can be implicitly represented by a sequence $I(v)$ of pairs $(side_j, size_j) \in \{L, R\} \times \mathbb{Z}^+$. Specifically, suppose the sequence $C(v)$ consists of h blocks $\ell_1, \ldots, \ell_h$ of size $x_1, \ldots, x_h$ interleaved with k blocks $r_1, \ldots, r_k$ of size $y_1, \ldots, y_k$, where $\ell_1 \cdots \ell_h$ is a prefix of the sequence $C(left(v))$ at the left child of v and $r_1 \cdots r_h$ is a prefix of the sequence $C(right(v))$ at the right child of v. Then $I(v)$ is an interleaving of the two sequences of pairs $[(L, x_1), \ldots, (L, x_h)]$ and $[(R, y_1), \ldots, (R, y_k)]$, where $(side_j, size_j) = (L, x_i)$, if the j'th block of $C(v)$ is ℓ_i, and $(side_j, size_j) = (R, y_i)$ if the j'th block of $C(v)$ is r_i. Note that $NL(v) = x_1 + \cdots + x_h$ and $NR(v) = y_1 + \cdots + y_k$. For example, if

$$\begin{aligned} C(left(v)) &= [5, 3, 1, 2], \\ C(right(v)) &= [9, 10, 15, 12, 15], \text{ and} \\ I(v) &= [(L, 1), (R, 3), (L, 3)], \end{aligned}$$

then $C(v) = [5, 9, 10, 15, 3, 1, 2]$, with $h = 2$, $\ell_1 = [5]$, $x_1 = 1$, $\ell_2 = [3, 1, 2]$, $x_2 = 3$, $k = 1$, $r_1 = [9, 10, 15]$, and $y_1 = 3$. If two consecutive blocks of $I(v)$ have the same side, they can be combined into one block, whose size is the sum of the sizes of those two blocks, without changing the sequence $C(v)$ it represents. Thus, we may assume, without loss of generality, that the sides of the blocks in $I(v)$ alternate between L and R.

Each internal node v has an LL/SC object $v.T$ containing a pointer into a persistent data structure T_v representing versions of the sequence $I(v)$ and, hence implicitly, the sequence $C(v)$. This data structure supports one update operation and two query operations. Here t is a pointer into T_v that indicates one version I of $I(v)$.

APPEND(t, x, y): return a pointer to a new version of $I(v)$ obtained from I by appending the pairs (L, x) and (R, y) to it, as appropriate. More specifically, if the last block of I is (L, z), update that block to $(L, z + x)$ and, if $y \neq 0$, append the pair (R, y). If the last block of I is (R, z), update that block to $(R, z + y)$ and, if $x \neq 0$, append the pair (L, x). When I is empty, append (L, x), if $x \neq 0$, and append (R, y), if $y \neq 0$.

Note that, if I is nonempty when $APPEND(t, x, y)$ is called, then the new sequence either has the same length as I or length one greater. If there are two pairs to append, the pair from the same side as the last pair in I is appended first. Two query operations are also supported.

BLOCKSUM(t, s, j): among the first j blocks of I, return the sum of the sizes of those blocks with first component s, i.e. BLOCKSUM$(t, s, j) = \sum\{size_h \mid side_h = s \text{ and } 1 \leq h \leq j\}$,

FINDBLOCK(t, s, m): return the minimum j with BLOCKSUM$(t, s, j) \geq m$.

Local variables:
v: a node in τ
$s \neq s'$: elements of $\{L, R\}$
t, t': pointers to nodes in T_v
j, m, h, k: nonnegative integers.

```
1    $m \leftarrow \text{READ}(N_i) + 1$
2    WRITE $N_i \leftarrow m$
3    $v \leftarrow$ process $i$'s leaf (in $\tau$)
4    while $v \neq \text{root}(\tau)$ do
5            if $v = left(parent(v))$
6            then $s \leftarrow L$
7                 $s' \leftarrow R$
8            else $s \leftarrow R$
9                 $s' \leftarrow L$
10           $v \leftarrow parent(v)$
             %$t$ is a pointer to the current root of $T_v$
11           $t \leftarrow \text{LL}(v.T)$
             %Check whether $i$'s instance of FI has reached $v$,
             %i.e. $C(v)$ contains at least $m$ elements from side $s$
12           while $\text{READ}(t.Ns) < m$ do
                     %Compute the length $h$ of $C(left(v))$
13                   if $left(v)$ is a leaf of $\tau$
14                   then $h \leftarrow \text{READ}(N_j)$, where $j$ is the index of this leaf
15                   else $t' \leftarrow \text{READ}(left(v).T)$
16                        $h \leftarrow \text{READ}(t'.NL) + \text{READ}(t'.NR)$
                     %Compute the length $k$ of $C(right(v))$
17                   if $right(v)$ is a leaf of $\tau$
18                   then $k \leftarrow \text{READ}(N_j)$, where $j$ is the index of this leaf
19                   else $t' \leftarrow \text{READ}(right(v).T)$
20                        $k \leftarrow \text{READ}(t'.NL) + \text{READ}(t'.NR)$
                     %Compute a pointer $t'$ to an updated version $T'$ of $T_v$
21                   $t' \leftarrow \text{APPEND}(t, h - t.NL, k - t.NR)$
                     %Try to update $v.T$ to point to $T'$
22                   $\text{SC}(v.T, t')$
23                   $t \leftarrow \text{LL}(v.T)$
             end while
             %Compute the position of $i$'s current instance in $C(v)$
             %by finding the block $j$ that contains the $m$'th element
             %from side $s$ and the sum of all previous blocks with side $s'$
24           $j \leftarrow \text{FINDBLOCK}(t, s, m)$
25           $m \leftarrow m + \text{BLOCKSUM}(t, s', j - 1)$
     end while
26   return $m - 1$
```

Fig. 1. Algorithm for FI performed by process i

Initially, $N_i = 0$, for $i = 1, \ldots, p$, and the sequences represented at every node are empty. Pseudocode for FI appears in Figure 1.

A persistent augmented balanced binary tree [9,8], such as a red-black tree or an AVL tree, is used to implement T_v. Each pair in the sequence $I(v)$ is represented by a node containing the side and the size of the pair. Nodes also contain pointers to their left and right children and balance information. They do not contain parent pointers. Each node u of T_v is augmented with the number of nodes in its subtree, the sum $u.NL$ of the sizes of the pairs in its subtree that have side L, and the sum $u.NR$ of the sizes of the pairs in its subtree that have side R. In particular, if T_v is nonempty, then $v.T$ is an LL/SC object that points to the root of a tree in T_v representing the sequence $I(v)$, $root(T_v).NL$ stores $NL(v)$, and $root(T_v).NR$ stores $NR(v)$. Initially, $v.T = nil$ and $NL(v) = NR(v) = 0$.

Processes do not change any information in nodes of T_v once they have been added to the data structure. Instead, when performing APPEND, they create new nodes containing the updated information. Thus, all of the ancestors of a changed node must also be changed. Although APPEND changes T_v, it does not affect $I(v)$ until $v.T$ is changed to the pointer it returns.

If $t = nil$, APPEND(t, x, y) creates a new tree containing (L, x), if $x \neq 0$, and (R, y), if $y \neq 0$. If $t \neq nil$, then APPEND(t, x, y) starts at the root in T_v pointed to by t and follows the rightmost path of its tree, making a copy of each node it encounters and pushing the copy onto a stack. If the side of the rightmost node in the tree is L, then x is added to its size and, if $y \neq 0$, the right child pointer of this node is changed from NIL to a new leaf that contains the element (R, y). Otherwise, y is added to its size and, if $x \neq 0$, the right child pointer of this node is changed from NIL to a new leaf that contains the element (L, x). Then, the stack is popped to progress back up the tree. As each node is popped, its right pointer is set to the root of the updated subtree. The information at the node, including its balance, is updated and rotations are performed, if necessary. The step complexity of APPEND is logarithmic in the number of nodes reachable from the root pointed to by t.

To perform a query operation, it suffices to perform the query as one would in the underlying augmented balanced binary tree, starting from a root. However, since the tree reachable from this root never changes while it can be accessed, there are no conflicts with update operations. Using an augmented, balanced binary tree to represent each version of $I(v)$ enables each query to be performed in time logarithmic in the length of the version to which it is applied.

Theorem 1. *A wait-free, linearizable, unbounded* FETCH&INC *object shared by p processes on which at most n* FI *operations are performed can be implemented so that each* FI *takes $O(\log p \log n)$ steps and each* READ *takes $O(1)$ steps.*

Proof (sketch). An instance of FI is linearized when $root(\tau).T$ is first updated during an APPEND (not necessarily performed by the same process) to point to the root a tree that contains information about this instance. At each node v of τ, the length of the sequence represented by any tree in T_v is at most n, so each operation on T_v can be performed in $O(\log n)$ steps. Since the tree τ has height $\Theta(\log p)$ and a process performs only a constant number of operations at each node on the path from its leaf to the root during an instance of FI, each FI operation takes $O(\log p \log n)$ steps.

To READ the value of the FETCH&INC object, a process reads $root(\tau).T$ to get a pointer to the current root of the tree representing $I(root(\tau))$. The READ is linearized at this step. If t is NIL, then the FETCH&INC object has its initial value, 0. Otherwise, its value is the sum of the persistent values $t.NL$ and $t.NR$, which is the length of $C(root(\tau))$ at the linearization point. This takes a constant number of steps. ■

Initially, this implementation uses $\Theta(p)$ space. Each FI operation adds $O(\log n)$ nodes to the data structure T_v, for each node v on the path from some leaf of τ to its root. Since τ has height $O(\log p)$, the total space used by this implementation to perform n operations is $O(p + n \log n \log p)$.

For a one-shot FETCH&INC object, $n \leq p$, so $O(\log^2 p)$ steps are used to perform each instance of FI and $O(p \log^2 p)$ registers and LL/SC objects are used.

3 The Second Implementation

We now present a more efficient implementation, which is obtained by compressing the tree in the data structure T_v that is pointed to by each node v of τ. Specifically, if there are $\ell = \ell(v)$ leaves in the subtree of τ rooted at v, we show how to ensure that the number of nodes reachable from each root of T_v is $O(\ell^2)$. This results in an implementation whose worst-case step complexity is $O(\log^2 p)$.

If $C(v) = [c_0, \ldots, c_{k-1}]$, we define $Q(v) = \{(j, c_j) \mid j = 0, \ldots, k-1\}$ to be the set of all position-id pairs. We say that a position j is *old at* v if there exists $i \in P(v)$ and $j' > j$ such that $(j, i), (j', i) \in Q(v)$, i.e., some id i occurs at position j in $C(v)$, but this is not the last occurrence of i. A position is *current at* v, if it is not old at v.

We take advantage of the fact that once a position becomes old at v, the identifier at that position is no longer accessed by any process. Thus, the identifiers at old positions can be permuted without affecting the outcome of pending or future FI operations. For example, if $I(v)$ contains three consecutive blocks (L, x), (R, y), (L, z), which represent $x + y + z$ old positions in $C(v)$, then we can replace these blocks with two blocks $(L, x+z), (R, y)$. Because the permuted sequence has fewer blocks, it can be represented by a tree with fewer nodes. An algorithm to compress ℓ consecutive positions of $C(v)$ is presented in Section 3.1. It has $O(\log \ell)$ step complexity.

We add a *deletion structure* $\Delta(v)$ at v to facilitate the identification of sequences of ℓ consecutive old positions to compress. It contains an array of $2\ell + 1$ *status-units*, which are described in Section 3.2. Each status-unit is associated with ℓ consecutive positions of the sequence $C(v)$. A persistent balanced binary search tree, A_v, enables processes to find the status-unit associated with any current position. When a position j becomes old, the process whose id is at position j of $C(v)$ records that fact in the status-unit associated with position j. The status-unit is also used to determine when all of its associated positions are old. A fixed *deletion tree*, D_v, with $2\ell+1$ leaves, described in Section 3.4, is used to keep track of such status units.

After all the positions associated with a status-unit have been compressed, the status-unit is recycled. This means that it is reinitialized and associated with a new sequence of positions. This is described in Section 3.5.

To perform FI, a process proceeds up the tree τ, as in the first implementation, starting from its leaf. Before propagating information up to a node v from its children, process i finds the status-unit associated with the position j in $C(v)$ that contains the id i added to the sequence when i last performed FI. Then it *marks* position j in that status-unit, to indicate that the position is old at v. If there is a status-unit whose associated positions are all old, process i also tries to compress these positions in the tree in T_v rooted at $v.T$ and recycle the status unit. The algorithm for performing FI is described in more detail in Section 3.6.

3.1 Compression

The idea of compression is as follows: Once all positions in $C(v)$ that correspond to a block (s,x), $s \in \{L,R\}$, of $I(v)$ are marked, the entire block can be marked by changing its side s to $s' \in \{L',R'\}$, i.e., by replacing (L,x) with (L',x) or (R,x) with (R',x). A block (s',x) with $s \in \{L',R'\}$ is called a *marked block*. Two adjacent marked blocks (s',y) and (s',z) with the same side s' can be replaced by a single marked block $(s',y+z)$. A sequence of consecutive marked blocks, $(s'_j,x_j),(s'_{j+1},x_{j+1}),\ldots,(s'_k,x_k)$, containing at least one with side L' and one with side R' can be replaced by two marked blocks (L',y) and (R',z), where $y=\sum\{x_i \mid j \le i \le k \text{ and } s'_i = L'\}$ and $z=\sum\{x_i \mid j \le i \le k \text{ and } s'_i = R'\}$. This is equivalent to permuting the elements in the corresponding locations of $C(v)$.

Suppose t is a pointer into T_v indicating a version I of $I(v)$ and suppose the ℓ consecutive positions $m,\ldots,m+\ell-1$ in a status-unit are all marked. These positions in I are compressed by updating T_v as follows: We assume $m>0$; the special case $m=0$ can be handled analogously. First, FINDBLOCK(t,m) and FINDBLOCK$(t,m+\ell+1)$ are used to find the blocks (s_j,x_j) and (s_k,x_k) that represent positions $m-1$ and $m+\ell$, respectively.

If $j=k$, then (s_j,x_j) is partitioned into three blocks, (s_j,x'_j), (s'_j,ℓ), and (s_j,x''_j), where x'_j is the number of positions less than m represented by (s_j,x_j) and x''_j is the number of positions greater than or equal to $m+\ell$ represented by (s_j,x_j). Note that block (s'_j,ℓ) is now marked, and all positions represented by that block are old. Moreover, the number of blocks in $I(v)$ and, hence, the number of nodes in T_v increased by 2.

Now suppose that $j \ne k$. If $j<k-1$, then all of the positions represented by blocks (s_i,x_i), $j<i<k$, are marked. These blocks are removed from the tree rooted at t.

If m is represented by block (s_j,x_j), then (s_j,x_j) is conceptually partitioned into two blocks (s_j,x'_j) and (s'_j,x''_j), where x'_j is the number of positions less than m represented by (s_j,x_j) and x''_j is the number of positions greater than or equal to m represented by (s_j,x_j). The block (s'_j,x''_j) is removed. This is accomplished by changing x_j to x'_j. Similarly, if $m+\ell-1$ is represented by block (s_k, x_k), then (s_k, x_k) is conceptually partitioned into two blocks (s'_k, x'_k) and (s_k, x''_k) where x'_k is the number of values less than $m+\ell$ represented by (s_k, x_k) and x''_k is the number of values greater than or equal to $m+\ell$, represented by (s_k, x_k). The block (s'_k, x'_k) is also removed.

If block (s_j, x_j) is marked and does not represent m, then it is removed and, if it is immediately preceded by a marked block, that block is removed, too. Similarly, if block (s_k, x_k) is marked and does not represent $m+\ell-1$, then it is removed, together with the next block, if it exists and is also marked. Note that, if there is a block immediately preceding or immediately following the removed blocks, it is unmarked.

Let x be the sum of the sizes of all the removed blocks with *side* $= L$. This can be computed in $O(\log \ell)$ steps directly from the tree rooted at t in T_v as the blocks are removed, using the augmented information at each node.

Similarly, let y be the sum of the sizes of all removed blocks with *side* $= R$. Finally, in place of the removed blocks, add the new marked block (L', x), if $x > 0$, and the new marked block (R', y), if $y > 0$. This maintains the invariant that there is at least one unmarked block between any two marked blocks with the same side. Since $I(v)$ contains $O(\ell^2)$ unmarked blocks, $I(v)$ contains $O(\ell^2)$ blocks in total. Hence, the tree in T_v that represents $I(v)$ has $O(\ell^2)$ nodes.

3.2 Status-Units

The deletion structure $\Delta(v)$ contains a collection of $2\ell+1$ *status-units*, $S_v[j]$, for $1 \leq j \leq 2\ell+1$. A status-unit has three parts: a *name*, a *flag*, and a *progress tree*. A name is a non-negative integer that can increase during an execution. When its name is g, the status-unit is associated with the ℓ consecutive positions $\ell g, \ldots, \ell g+\ell-1$. Initially, status-unit $S_v[j]$ has name $j-1$, for $j = 1, \ldots, 2\ell+1$.

An LL/SC object can be used to store the name of a status unit. However, the name of a status unit grows each time it is recycled. To avoid using large LL/SC objects, we represent the name of a status unit using an LL/SC object *namer*, which stores a process identifier in $P(v)$, and an array, *names*, of ℓ single-writer registers, indexed by $P(v)$. At any time, the name of the status unit is the value of *names*[*namer*].

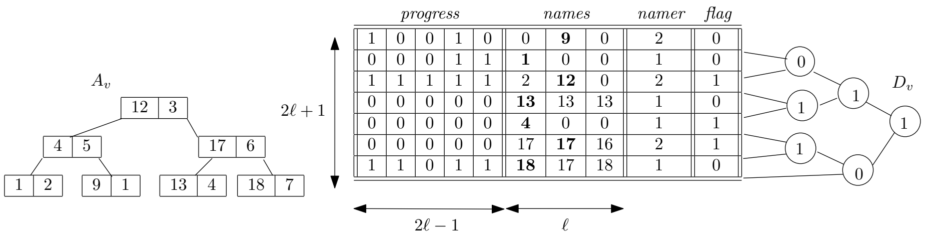
The *flag* is a single-bit LL/SC object. It is initially 0 and it is reset to 0 whenever the status-unit changes its name. After all of its associated positions have been marked, its *flag* is changed from 0 to 1. This indicates that these positions can be compressed. After that has been done, the status-unit can be reused.

The *progress tree* is a fixed full binary tree on ℓ leaves, represented implicitly by an array *progress*$[1..2\ell-1]$ of $\ell-1$ single-bit LL/SC objects and ℓ single-bit registers. It enables processes to determine when all the positions, $\ell g, \ldots, \ell g + \ell-1$, represented by a status-unit with name g are old at v. Progress trees were introduced for processes to keep track of their collective progress performing a collection of tasks [4,7].

When a status-unit is reused, its progress tree also needs to be reused. Because it has $2\ell-1$ fields, it would take too much time to reinitialize all of them to 0.

Thus, we need an implementation of a progress tree that can be reused without being reinitialized. This is discussed in Section 3.3.

Each node of the tree A_v stores the name of one status-unit. The index of the status-unit with that name is stored as auxiliary data. Processes make updates to A_v similarly to the way they make updates to T_v. Initially, A_v stores the initial names of all $2\ell + 1$ status-units (i.e., name $j - 1$ and auxiliary data j for status-unit j, $1 \leq j \leq 2\ell + 1$).



progress					*names*			*namer*	*flag*
1	0	0	1	0	0	**9**	0	2	0
0	0	0	1	1	**1**	0	0	1	0
1	1	1	1	1	2	**12**	0	2	1
0	0	0	0	0	**13**	13	13	1	0
0	0	0	0	0	**4**	0	0	1	1
0	0	0	0	0	17	**17**	16	2	1
1	1	0	1	1	**18**	17	18	1	0

Fig. 2. An example of a deletion structure $\Delta(v)$ for $\ell = 3$

Figure 2 presents an example of a deletion structure for a node v of τ with $\ell = 3$ leaves in its subtree. The j'th row in the table represents the status-unit $S_v[j]$, for $j = 1, \ldots, 7 = 2\ell + 1$. Its name is indicated in bold. The tree on the left is A_v and the tree on the right is D_v.

3.3 Reusable Progress Trees

A reusable progress tree is represented implicitly using an array of length $2\ell - 1$ (as in a binary heap). We will use *leaf*(m) to denote the location of the m'th leaf in this array and use $parent(u)$, $left(u)$, and $right(u)$ to represent the locations of the parent, left child, and right child, respectively, of node u.

Each node of the progress tree stores a single bit. When the $flag$ of a status-unit is 1, all bits of its progress tree are the same. Before any nodes in the progress tree are changed, the $flag$ is reset to 0. Only the process that received position $g\ell + m - 1$ can change the value of *leaf*(m) in the progress tree of the status-unit with name g. Say it changes this bit from $1 - b$ to b. After doing so, the process progresses up the tree, setting the bit at each ancestor of *leaf*(m) to b, if the bits at both of the children of that ancestor are b. Thus, an internal node of the tree is changed only after all of the leaves in its subtree have been changed. When the bit at the root of the progress tree changes to b, all of the bits in the tree are b and the $flag$ can be changed from 0 to 1. After that, the $flag$ is not reset to 0 again until *namer* is changed.

A process that is progressing up the tree, but falls asleep for a long time, should not change bits in the *progress* tree of a status-unit that has since changed its name. To ensure this, each bit corresponding to a non-leaf node of the *progress* tree is an LL/SC object. Before changing its leaf, the process performs LL(*namer*). To change the bit at an internal node u to b, the process performs LL(u) followed by VL($namer$) and only performs SC(u, b) and continues to $parent(u)$ if the validation indicates that $namer$ has not been updated. If the validation is unsuccessful, the process is done, since the bit at the root has already been changed to b. After a process changes the bit at the root of the progress tree to b, it performs LL($flag$) and VL($namer$) to verify that $flag$ has value 0 and $namer$ has not changed. If successful, it then performs SC($flag, 1$) to change $flag$ to 1.

Pseudocode is presented on lines 1–10 of Figure 3.

```
1   $u \leftarrow leaf(m)$
2   $b \leftarrow \neg S_v[j].progress[u]$
3   LL($S_v[j].namer$)
    %Change the mark at $leaf(m)$
4   $S_v[j].progress[u] \leftarrow b$
5   while $u \neq root$ do
6       if $S_v[j].progress[sibling(u)] = \neg b$ then return
7       $u \leftarrow parent(u)$
8       LL($S_v[j].progress[u]$)
9       if $\neg$ VL($S_v[j].namer$) then return
10      SC($S_v[j].progress[u], b$)
    end while
11  LL($S_v[j].flag$)
12  if $\neg$ VL($S_v[j].namer$) then return
13  if $\neg$ SC($S_v[j].flag, 1$) then return
14  $u \leftarrow j$'th leaf of $D_v$
15  while $u \neq root$ do
16      $u \leftarrow parent(u)$
17      LL($D_v[u]$)
18      if $\neg$ VL($S_v[j].namer$) then return
19      if SC($D_v[u], 1$) = false
20      then if LL($D_v[u]$) = 0
21          then if $\neg$ VL($S_v[j].namer$) then return
22              SC($D_v[u], 1$)
    end while
23  return
```

Fig. 3. Algorithm to mark position $\ell n' + m - 1$ in status-unit $S_v[j]$ with name n' and add j to D_v, if necessary

3.4 Deletion Tree D_v

The deletion tree D_v is used to represent the marked status-units, i.e. whose flags are set. Hence, the positions associated with these status-units can be compressed. This data structure allows a process to efficiently find such a status-unit. It can also be updated efficiently. It is a fixed full binary tree whose leaves are $S_v[1].flag, \ldots, S_v[2\ell+1].flag$. Each non-leaf node is a single-bit LL/SC object, which is initially 0.

When a process changes the flag of the status unit $S_v[j]$ from 0 to 1, it adds j to the set by walking up the tree D_v starting from the parent of this leaf, trying to set every LL/SC object it visits on this path to 1. Because the process may be slow or may fall asleep for a long time, status-unit $S_v[j]$ may be reallocated before the process reaches the root of D_v. To prevent this from causing problems, the process proceeds as in a reusable progress tree: For each node u that the process visits on the path it first executes LL($D_v[u]$), then VL($S_v[j].namer$) and finally performs SC($D_v[u], 1$) if and only if the validation of $S_v[j].namer$ was successful. If the validation was successful, but the SC fails and $D_v[u]$ is still 0 after the unsuccessful SC, the process repeats the LL and VL a second time and, if the validation is now successful, it also performs SC a second time. If the validations are successful, the process proceeds to $parent(u)$. If any validation is unsuccessful, $S_v[j]$ has already been recycled and the process does not continue. Pseudocode appears on lines 11–23 of Figure 3.

While a status-unit $S_v[j]$ is being recycled, its flag gets reset to 0, and j has to be removed from D_v. To do so, a process proceeds up the tree D_v on the path from the i'th leaf to the root, trying to reset the bit at each node to 0 until it finds a node which has a child with value 1, indicating the presence of a leaf with value 1 in its subtree. Specifically, at the non-leaf node u, the process performs LL($D_v[u]$) and, if 0 is returned, it proceeds to $parent(u)$ (or is done, if u is the root). If u was 1, then the process performs LL($D_v[left(u)]$) and LL($D_v[right(u)]$). If at least one of the LL operations returns 1, the process is done. Otherwise, it performs VL(v). If the validation is unsuccessful, the process is done. Otherwise, it performs SC($D_v[u], 0$). If $D_v[u]$ is still 1 after the SC, the process repeats the LL's follows by a VL a second time and, if no child of u has value 1 and the validation is successful, the process also performs SC a second time. If u is 0 after either of these SC's, the process continues to $parent(u)$ (or is done, if u is the root). If not, some other process adding some value j into D_v performed LL($D_v[u]$) and SC($D_v[u], 1$) between the first process's first LL($D_v[u]$) and its last SC($D_v[u], 0$). In this case, $S_v[j].flag$ is a leaf of node u and had value 1 between the LL and SC, and the first process can stop.

More generally, the following invariants will be maintained:

— if a non-leaf node u of D_v is 0, then either there are no leaves with value 1 in the subtree of D_v rooted at u or there is some leaf in its subtree that has value 1 and the process that last changed this leaf to 1 is at or below node u, and

— if a non-leaf node u of D_v is 1, then there is a leaf in the subtree of D_v rooted at u that either has value 1 or is being recycled by some process that is at or below node u.

Note that multiple status-units can be added to D_v concurrently, but only one status unit is removed from D_v at a time.

To find a marked status-unit (i.e., one whose flag is set), a process walks down the tree from the root to a leaf, at each node reading the values of its children, and proceeding to a child whose value is 1, if such a child exists. If the process reaches a leaf j with value 1, this means that $S_v[j].flag$ is set. It may happen that the process gets stuck at a node whose children both have value 0, in which case the process aborts its attempt to find a marked status-unit.

3.5 Recycling

Status units are recycled one at a time. The node v has a field $v.E$ that indicates which status unit is to be recycled next. When a process recycles a status unit $S_v[j]$, it first tries to change its name. Process i begins by writing a proposed new name into the single-writer register $S_v[j].names[i]$. Then it tries to change $S_v[j].namer$ to i by performing $\text{LL}(S_v[j].namer)$ followed by $\text{SC}(S_v[j].namer, i)$. If the SC is successful, this changes the name of $S_v[j]$ to the name it proposed. To prevent a slow process from accidentally changing the name of a status-unit that has already been recycled, process i performs a VL at v between the LL and the SC. If the validation is unsuccessful, then the recycling of status-unit $S_v[j]$ has already been completed and process i does not continue trying to recycle it.

Next, process i tries to change $S_v[j].flag$ from 1 to 0 by executing $\text{LL}(S_v[j].flag)$ and, if that returns 1, performing $\text{SC}(S_v[j].flag, 0)$. Again, process i performs a VL at v between each LL and matching SC to see whether $S_v[j]$ has finished being recycling and, if so, does not continue to try to recycle it. Then process i removes j from the set represented by D_v, as described in Section 3.4.

3.6 Overall Algorithm

Instead of the LL/SC object $v.T$, as in the first implementation, we now have an LL/SC object with three fields, $v.T$, $v.A$, and $v.E$. To avoid having an LL/SC object with multiple fields, we could use indirection and, instead, have the LL/SC object contain a pointer to a record with three registers.

The first two fields contain (pointers to) the trees of T_v and A_v, rooted at $v.T$ and $v.A$, respectively. The last field contains an element e of $\{0, \ldots, 2\ell + 1\}$. When $e \neq 0$, $S_v[e]$ is a status unit that is ready to be recycled, i.e. it has been marked, the positions associated with its current name have been compressed, and there is no node in A_v whose key is this name.

The algorithm to perform FI is a modification of the algorithm in Figure 1. Lines 11 and 23 are replaced by $(t, a, e) \leftarrow \text{LL}(v.T, v.A, v.E)$. Since T_v and A_v are persistent data structures, the trees rooted at t and a do not change during an iteration of the while loop beginning on line 12.

Before the body of this loop is performed, process i finds the largest name n'' stored in the tree rooted at a. If process i has previously performed an instance of FI, it determines the name of the status-unit that is associated with the last position m' of i in $C(v)$. Then it searches in the binary search tree rooted at a for the key with this name. Since position m' is not yet marked, it can be shown that a node having this name will be found. Let j' be the index of the status-unit, which is also stored in this node. Process i marks m' as old in $S_v[j']$ and propagates this change up the progress tree, as described in Section 3.3.

If $e \neq 0$, then process i tries to recycle the status-unit $S_v[e]$ by updating its name to $n'' + 1$, changing $S_v[e].flag$ from 1 to 0, and deleting e from the set represented by D_v, as described in Section 3.5.

Next, process i tries to find a status-unit whose flag is 1, using the deletion tree D_v as described in Section 3.4. If it finds such a status unit e', then process i compresses the positions associated with the status-unit $S_v[e']$ in the tree rooted at t, as described in Section 3.1, and uses the root of the resulting tree in place of t in line 21. Otherwise, $e' = 0$.

If $e' = e = 0$, let $a' = a$. Otherwise, process i creates a new tree in A_v starting from the tree with root a by adding a node with key $n'' + 1$ and auxiliary data e (if $e \neq 0$) and removing the node with key $S_v[e'].names[S_v[e'].namer]$ (if $e' \neq 0$). Let a' denote the root of this tree.

Finally, line 22 in Figure 1 is replaced with SC$((v.T, v.A, v.E), (t', a', e'))$, where t' is the result computed on line 21. Note that unless this SC is successful, process i makes no modifications to the trees of T_v and A_v, rooted at $v.T$ and $v.A$, respectively. However, the changes made by each process to status-units and D_v occur asynchronously before it attempts this SC at the end of the iteration.

At any point in time, at most 2ℓ of the positions in $\{0, \ldots, |C(v)| - 1\}$ are not marked: the current position for each process and possibly its previous position. Since each status-unit is associated with ℓ positions, there are $O(\ell^2)$ positions represented in the uncompressed portion of the tree in T_v rooted at $v.T$. It follows that this tree has $O(\ell^2)$ nodes. We now state our main theorem.

Theorem 2. *A wait-free, linearizable, unbounded* FETCH&INC *object shared by p processes can be implemented so that each* FETCH&INC *takes $O(\log^2 p)$ steps and each READ takes $O(1)$ steps, regardless of the number of* FETCH&INC *operations performed. Assuming garbage collection is performed, the number of registers and LL/SC objects needed is $O(p^2)$.*

Proof (sketch). The linearization points of FI and READ are the same as in the proof of Theorem 1. Suppose v is a node of τ with $\ell \leq p$ leaves in its subtree. Since $O(\ell)$ and $O(\ell^2)$ nodes are reachable from any root of a tree in A_v and T_v, respectively, searches and updates in these persistent data structures take $O(\log p)$ steps. Likewise, D_v and the progress trees are fixed balanced trees with $2\ell + 1$ and ℓ leaves, respectively, so operations on them also take $O(\log p)$ steps. Since τ has height $O(\log p)$, it follows that FI has $O(\log^2 p)$ step complexity. As in the first implementation, READ can be performed in a constant number of steps.

Excluding A_v, $O(\ell^2)$ registers and LL/SC objects are used to represent $\Delta(v)$. There are $O(\ell^2)$ nodes, each consisting of a constant number of objects, reachable from the roots of A_v and T_v. Summing over all internal nodes v of τ gives a total of $O\left(\sum_{j=0}^{\lceil \log_2 p \rceil - 1} 2^j (2^{\lceil \log_2 p \rceil - j})^2\right) = O(p^2)$ objects.

Although only a bounded number of nodes in A_v and T_v are reachable from $v.A$ and $v.T$, the total number of nodes in these persistent data structures grows with n. However, full persistence is not needed by our implementation: When a node is no longer reachable from v or the local pointer of any process, it can be removed to save space, since it will not be accessed from then on. At any point in time, each process has a constant number of local pointers into persistent structures and $O(p^2)$ nodes are reachable from each of them. Thus, the total number of registers and LL/SC objects used by our implementation is $O(p^3)$, assuming garbage collection is performed.

4 Extensions

Our implementations of Fetch&Inc can be extended to Fetch&Add by having each element of the sequence $C(v)$ contain the input to each instance of fa, together with the identifier of the process that performed the instance. Likewise, each block of $I(v)$ can be augmented with the sum of the inputs to all instances occurring in or before this block and that are from the same side. Details of this algorithm and a proof of its correctness will appear in the full version of the paper. Ways to remove nodes from A_v and T_v that are no longer reachable will also be addressed.

References

1. Afek, Y., Dauber, D., Touitou, D.: Wait-free made fast. In: Proc. of 27th ACM STOC, pp. 538–547 (1995)
2. Afek, Y., Weisberger, E., Weisman, H.: A completeness theorem for a class of synchronization objects. In: Proc. of 12th PODC, pp. 159–170 (1993)
3. Alistarh, D., Aspnes, J., Censor-Hillel, K., Gilbert, S., Zadimoghaddam, M.: Optimal-time adaptive strong renaming, with applications to counting. In: Proc. of 30th PODC, pp. 239–248 (2011)
4. Anderson, R.J., Woll, H.: Algorithms for the certified write-all problem. SIAM J. Comput. 26(5), 1277–1283 (1997)
5. Anderson, T.: The performance of spin lock alternatives for shared-money multiprocessors. IEEE Trans. Parallel Distrib. Syst. 1(1), 6–16 (1990)
6. Aspnes, J., Herlihy, M., Shavit, N.: Counting networks. J. of the ACM 41(5), 1020–1048 (1994)
7. Buss, J.F., Kanellakis, P.C., Ragde, P., Shvartsman, A.A.: Parallel algorithms with processor failures and delays. J. Algorithms 20(1), 45–86 (1996)
8. Clements, A.T., Kaashoek, M.F., Zeldovich, N.: Scalable address spaces using RCU balanced trees. In: 17th ASPLOS, pp. 199–210 (2012)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press (2001)
10. Freudenthal, E., Gottlieb, A.: Process coordination with fetch-and-increment. In: Proc. of ASPLOS-IV, pp. 260–268 (1991)
11. Goodman, J., Vernon, M., Woest, P.: Efficent synchronization primitives for large-scale cache-coherent multiprocessors. In: Proc. of ASPLOS-III, pp. 64–75 (1989)
12. Gottlieb, A., Lubachevsky, B., Rudolph, L.: Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. ACM Trans. Program. Lang. Syst. 5(2), 164–189 (1983)
13. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991)
14. Herlihy, M.: A methodology for implementing highly concurrent objects. ACM Trans. Program. Lang. Syst. 15(5), 745–770 (1993)
15. Herlihy, M., Shavit, N., Waarts, O.: Linearizable counting networks. Distr. Comp. 9(4), 193–203 (1996)
16. Jayanti, P.: A time complexity lower bound for randomized implementations of some shared objects. In: Proc. of 17th PODC, pp. 201–210 (1998)
17. Jayanti, P., Tan, K., Toueg, S.: Time and space lower bounds for nonblocking implementations. SIAM J. Comput. 30(2), 438–456 (2000)

Show No Weakness: Sequentially Consistent Specifications of TSO Libraries

Alexey Gotsman[1], Madanlal Musuvathi[2], and Hongseok Yang[3]

[1] IMDEA Software Institute
[2] Microsoft Research
[3] University of Oxford

Abstract. Modern programming languages, such as C++ and Java, provide a sequentially consistent (SC) memory model for well-behaved programs that follow a certain synchronisation discipline, e.g., for those that are data-race free (DRF). However, performance-critical libraries often violate the discipline by using low-level hardware primitives, which have a weaker semantics. In such scenarios, it is important for these libraries to protect their otherwise well-behaved clients from the weaker memory model.

In this paper, we demonstrate that a variant of linearizability can be used to reason formally about the interoperability between a high-level DRF client and a low-level library written for the Total Store Order (TSO) memory model, which is implemented by x86 processors. Namely, we present a notion of linearizability that relates a concrete library implementation running on TSO to an abstract specification running on an SC machine. A client of this library is said to be DRF if its SC executions calling the abstract library specification do not contain data races. We then show how to compile a DRF client to TSO such that it only exhibits SC behaviours, despite calling into a racy library.

1 Introduction

Modern programming languages, such as C++ [3,2] and Java [11], provide memory consistency models that are weaker than the classical *sequential consistency (SC)* [10]. Doing so enables these languages to support common compiler optimisations and to compile efficiently to modern architectures, which themselves do not guarantee SC. However, programming on such *weak memory models* can be subtle and error-prone. As a compromise between programmability and performance, C++ and Java provide *data-race free (DRF)* memory models, which guarantee SC for programs without data races, i.e., those that protect data accesses with an appropriate use of high-level synchronisation primitives defined in the language, such as locks and semaphores[1].

While DRF memory models protect most programmers from the counter-intuitive effects of weak memory models, performance-minded programmers often violate the DRF discipline by using low-level hardware primitives. For instance, it is common for a systems-level C++ program, such as an operating system kernel, to call into highly-optimised libraries written in assembly code. Moreover, the very synchronisation primitives of the high-level language that programmers use to ensure DRF are

[1] C++ [3,2] also includes special *weak atomic* operations that have a weak semantics. Thus, a C++ program is guaranteed to be SC only if it is DRF and avoids the use of weak atomics.

usually implemented in its run-time system in an architecture-specific way. Thus, it becomes necessary to reason about the interoperability between low-level libraries *native* to a particular hardware architecture and their clients written in a high-level language. While it is acceptable for expert library designers to deal with weak memory models, high-level language programmers need to be protected from the weak semantics.

In this paper, we consider this problem for libraries written for the *Total Store Order (TSO)* memory model, used by x86 processors (and described in Section 2). TSO allows for the *store buffer* optimisation implemented by most modern processors: writes performed by a processor are buffered in a processor-local store buffer and are flushed into the memory at some later time. This complicates the interoperability between a client and a TSO library. For instance, the client cannot assume that the effects of a library call have taken place by the time the call returns. Our main contributions are:

- a notion of specification of native TSO libraries in terms of the concepts of a high-level DRF model, allowing the model to be extended to accommodate such libraries, while preserving the SC semantics; and
- conditions that a compiler has to satisfy in order to implement the extended memory model correctly.

Our notion of library specification is based on *linearizability* [9], which fixes a correspondence between a *concrete* library and an *abstract* one, the latter usually implemented atomically and serving as a specification for the former. To reason formally about the interoperability between a high-level DRF client and a low-level TSO library, we propose a variant of linearizability called *TSO-to-SC linearizability* (Section 3). It relates a concrete library implementation running on the TSO memory model to its abstract specification running on SC. As such, the abstract specification describes the behaviour of the library in a way compatible with a DRF memory model. Instead of referring to hardware concepts, it fakes the effects of the concrete library implementation executing on TSO by adding extra non-determinism into SC executions. TSO-to-SC linearizability is compositional and allows soundly replacing a library by its SC specification in reasoning about its clients.

TSO-to-SC linearizability allows extending DRF models of high-level languages to programs using TSO libraries by defining the semantics of library calls using their SC specifications. In particular, this allows generalising the notion of data-race freedom to such programs: a client using a TSO library is DRF if so is every SC execution of the same client using the SC library specification. Building on this, we propose requirements that a compiler should satisfy in order to compile such a client onto a TSO machine correctly (Section 5), and establish the *Simulation Theorem* (Theorem 13, Section 5), which guarantees that a correctly compiled DRF client produces only SC behaviours, despite calling into a native TSO library. The key benefit of our framework is that both checking the DRF property of the client and checking the compiler correctness does not require TSO reasoning. Reasoning about weak memory is only needed to establish the TSO-to-SC linearizability of the library implementation. However, this also makes the proof of the Simulation Theorem challenging.

Our results make no difference between custom-made TSO libraries and TSO implementations of synchronisation primitives built into the run-time system of the high-level language. Hence, TSO-to-SC linearizability and the Simulation Theorem provide conditions ensuring that a given TSO implementation of the run-time system for a DRF language has the desired semantics and interacts correctly with its compilation.

Recently, a lot of attention has been devoted to criteria for checking whether a TSO program produces only sequentially consistent behaviours [12,4,1]. Such criteria are less flexible than TSO-to-SC linearizability, as they do not allow a program to have internal non-SC behaviours; however, they are easier to check. We therefore also analyse which of the criteria can be used for establishing the conditions required by our framework (Sections 4 and 6).

Proofs of all the theorems stated in the paper are given in [7, Appendix C].

2 TSO Semantics

Due to space constraints, we present the TSO memory model only informally; a formal semantics is given in [7, Appendix A]. The most intuitive way to explain TSO is using an abstract machine [13]. Namely, consider a multiprocessor with n CPUs, indexed by $\mathsf{CPUid} = \{1, \dots, \mathsf{NCPUs}\}$, and a shared memory. The state of the memory is described by an element of $\mathsf{Heap} = \mathsf{Loc} \rightharpoonup \mathsf{Val}$, where Loc and Val are unspecified sets of locations and values, such that $\mathsf{Loc} \subseteq \mathsf{Val}$. Each CPU has a set of general-purpose registers $\mathsf{Reg} = \{\mathtt{r}_1, \dots, \mathtt{r}_m\}$ storing values from Val. In TSO, processors do not write to memory directly. Instead, every CPU has a ***store buffer***, which holds write requests that were issued by the CPU, but have not yet been ***flushed*** into the shared memory. The state of a buffer is described by a sequence of location-value pairs.

The machine executes programs of the following form:

$$L ::= \{m = C_m \mid m \in M\} \qquad\qquad C(L) ::= \mathsf{let}\ L\ \mathsf{in}\ C_1 \parallel \dots \parallel C_{\mathsf{NCPUs}}$$

A program $C(L)$ consists of a declaration of a library L, implementing methods $m \in M \subseteq \mathsf{Method}$ by commands C_m, and its client, specifying a command C_t to be run by the (hardware) thread in each CPU t. For the above program we let $\mathsf{sig}(L) = M$. We assume that the program is stored separately from the memory. The particular syntax of commands C_t and C_m is of no concern for understanding the main results of this paper and is deferred to [7, Appendix A]. We consider programs using a single library for simplicity only; we discuss the treatment of multiple libraries in Section 3.

The abstract machine can perform the following transitions:

- A CPU wishing to write a value to a memory location adds an appropriate entry to the *tail* of its store buffer.
- The entry at the *head* of the store buffer of a CPU is flushed into the memory at a non-deterministically chosen time. Store buffers thus have the FIFO ordering.
- A CPU wishing to read from a memory location first looks at the pending writes in its store buffer. If there are entries for this location, it reads the value from the newest one; otherwise, it reads the value directly from the memory.
- Modern multiprocessors provide commands that can access several memory locations atomically, such as compare-and-swap (CAS). To model this in our machine, a CPU can execute a special lock command, which makes it the only CPU able to execute commands until it executes an unlock command. The unlock command

has a built-in ***memory barrier***, forcing the store buffer of the CPU executing it to be flushed completely. This can be used by the programmer to recover SC when needed.

– Finally, a CPU can execute a command affecting only its registers. In particular, it can call a library method or return from it (we disallow nested method calls).

The behaviour of programs running on TSO can sometimes be counter-intuitive. For example, consider two memory locations x and y initially holding 0. On TSO, if two CPUs respectively write 1 to x and y and then read from y and x, as in the following program, it is possible for both to read 0 in the same execution:

```
              x = y = 0;
x = 1; b = y;   ||   y = 1; a = x;
```

$$\{\mathtt{a} = \mathtt{b} = 0\}$$

Here a and b are local variables of the corresponding threads, stored in CPU registers. The outcome shown cannot happen on an SC machine, where both reads and writes access the memory directly. On TSO, it happens when the reads from y and x occur before the writes to them have propagated from the store buffers of the corresponding CPUs to the main memory. Note that executing the writes to x and y in the above program within lock..unlock blocks (which on x86 corresponds to adding memory barriers after them) would make it produce only SC behaviours.

We describe computations of the machine using ***traces***, which are finite sequences of ***actions*** of the form

$$\begin{aligned}\varphi \ ::= \ & (t, \mathsf{read}(x,u)) \mid (t, \mathsf{write}(x,u)) \mid (t, \mathsf{flush}(x,u)) \mid \\ & (t, \mathsf{lock}) \mid (t, \mathsf{unlock}) \mid (t, \mathsf{call}\ m(r)) \mid (t, \mathsf{ret}\ m(r))\end{aligned}$$

where $t \in \mathsf{CPUid}$, $x \in \mathsf{Loc}$, $u \in \mathsf{Val}$, $m \in \mathsf{Method}$ and $r \in \mathsf{Reg} \to \mathsf{Val}$. Here $(t, \mathsf{write}(x,u))$ corresponds to enqueuing a pending write of u to the location x into the store buffer of CPU t, $(t, \mathsf{flush}(x,u))$ to flushing a pending write of u to the location x from the store buffer of t into the shared memory. The rest of the actions have the expected meaning. Of transitions by a CPU affecting solely its registers, only calls and returns are recorded in traces. We assume that parameters and return values of library methods are passed via CPU registers, and thus record their values in call and return actions. We use the standard notation for traces: $\tau(i)$ is the i-th action in the trace τ, $|\tau|$ is its length, and $\tau|_t$ its projection to actions by CPU t. We denote the concatenation of two traces τ_1 and τ_2 with $\tau_1\tau_2$.

Given a suitable formalisation of the abstract machine transitions, we can define the set of traces $[\![C(L)]\!]_{\mathsf{TSO}}$ generated by executions of the program $C(L)$ on TSO [7, Appendix A]. For simplicity, we do not consider traces that have a (t, lock) action without a matching (t, unlock) action.

To give the semantics of a program on the SC memory model, we do not define another abstract machine; instead, we identify the SC executions of a program with those of the TSO machine that flush all writes immediately. Namely, we let $[\![C(L)]\!]_{\mathsf{SC}}$ be the set of sequentially consistent traces from $[\![C(L)]\!]_{\mathsf{TSO}}$, defined as follows.

DEFINITION 1. *A trace is* ***sequentially consistent (SC)****, if every action* $(t, \mathsf{write}(x,u))$ *in it is immediately followed by* $(t, \mathsf{flush}(x,u))$.

We assume that the set of memory locations Loc is partitioned into those owned by the client (CLoc) and the library (LLoc): $\mathsf{Loc} = \mathsf{CLoc} \uplus \mathsf{LLoc}$. The client C and the library L are ***non-interfering*** in $C(L)$, if in every computation from $[\![C(L)]\!]_{\mathsf{TSO}}$, commands performed by the client (library) code access only locations from CLoc (LLoc). In the following, we consider only programs where the client and the library are non-interfering. We provide pointers to lifting this restriction in Section 7.

3 TSO-to-SC Linearizability

We start by presenting our notion of library specification, discussing its properties and giving example specifications. The notion of specification forms the basis for interoperability conditions presented in Section 5.

TSO-to-SC Linearizability. When defining library specifications, we are not interested in internal library actions recorded in traces, but only in interactions of the library with its client. We record such interactions using ***histories***, which are traces including only ***interface actions*** of the form $(t, \mathsf{call}\ m(r))$ or $(t, \mathsf{ret}\ m(r))$, where $t \in \mathsf{CPUid}$, $m \in \mathsf{Method}$, $r \in \mathsf{Reg} \to \mathsf{Val}$. Recall that r records the values of registers of the CPU that calls the library method or returns from it, which serve as parameters or return values. We define the history $\mathsf{history}(\tau)$ of a trace τ as its projection to interface actions and lift history to sets T of traces pointwise: $\mathsf{history}(T) = \{\mathsf{history}(\tau) \mid \tau \in T\}$. In the following, we write _ for an expression whose value is irrelevant.

DEFINITION 2. *The **linearizability relation** is a binary relation $\sqsubseteq$ on histories defined as follows: $H \sqsubseteq H'$ if $\forall t \in \mathsf{CPUid}.\, H|_t = H'|_t$ and there is a bijection $\pi\colon \{1, \ldots, |H|\} \to \{1, \ldots, |H'|\}$ such that $\forall i.\, H(i) = H'(\pi(i))$ and $\forall i, j.\, i < j \wedge H(i) = (\_, \mathsf{ret}\ \_) \wedge H(j) = (\_, \mathsf{call}\ \_) \Rightarrow \pi(i) < \pi(j)$.*

That is, H' linearizes H when it is a permutation of the latter preserving the order of actions within threads and non-overlapping method invocations.

To generate the set of all histories of a given library L, we consider its ***most general client***, whose hardware threads on every CPU repeatedly invoke library methods in any order and with any parameters possible. Its formal definition is given in [7, Appendix B]. Informally, assume $\mathsf{sig}(L) = \{m_1, \ldots, m_l\}$. Then $\mathsf{MGC}(L) = (\mathsf{let}\ L\ \mathsf{in}\ C_1^{\mathsf{mgc}} \parallel \ldots \parallel C_{\mathsf{NCPUs}}^{\mathsf{mgc}})$, where for all t, the command C_t^{mgc} behaves as

```
while (true) { havoc; if (*) $m_1$; else if (*) $m_2$; ... else $m_l$; }
```

Here $*$ denotes non-deterministic choice, and `havoc` sets all registers storing method parameters to arbitrary values. The set of traces $[\![\mathsf{MGC}(L)]\!]_{\mathsf{TSO}}$ includes all library behaviours under any possible client. We write $[\![L]\!]_{\mathsf{TSO}}$ for $[\![\mathsf{MGC}(L)]\!]_{\mathsf{TSO}}$ and $[\![L]\!]_{\mathsf{SC}}$ for $[\![\mathsf{MGC}(L)]\!]_{\mathsf{SC}}$. We can now define what it means for a library executing on SC to be a specification for another library executing on TSO.

DEFINITION 3. *For libraries L_1 and L_2 such that $\mathsf{sig}(L_1) = \mathsf{sig}(L_2)$, we say that L_2 **TSO-to-SC linearizes** L_1, written $L_1 \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L_2$, if $\forall H_1 \in \mathsf{history}([\![L_1]\!]_{\mathsf{TSO}}).\, \exists H_2 \in \mathsf{history}([\![L_2]\!]_{\mathsf{SC}}).\, H_1 \sqsubseteq H_2$.*

```
word x=1;

void acquire()
{
  while(1) {
    lock;
    if (x==1) {
      x=0;
      unlock;
      return;
    }
    unlock;
    while(x==0);
  }
}

void release()
{
  x=1;
}

int tryacquire()
{
  lock;
  if (x==1) {
    x=0; unlock;
    return 1;
  }
  unlock;
  return 0;
}
```

(a)

```
word x=1;

void acquire()
{
  lock;
  assume(x==1);
  x=0;
  unlock;
}

void release()
{
  x=1;
}

int tryacquire()
{
  lock;
  if (x==1 && *)
  {
    x=0;
    unlock;
    return 1;
  }
  unlock;
  return 0;
}
```

(b)

Fig. 1. (a) L_{spinlock}: a test-and-test-and-set spinlock implementation on TSO; (b) $L^{\sharp}_{\mathsf{spinlock}}$: its SC specification. Here $*$ denotes non-deterministic choice. The `assume`(E) command acts as a filter on states, choosing only those where E evaluates to non-zero values (see [7, Appendix A]).

Thus, L_2 linearizes L_1 if every history of the latter on TSO may be reproduced in a linearized form by the former on SC. When the library L_2 is implemented atomically, and so histories in $\mathsf{history}(\llbracket L_2 \rrbracket_{\mathsf{SC}})$ are sequential, Definition 3 becomes identical to the standard linearizability [9], except the libraries run on different memory models.

Example: Spinlock. Figure 1a shows a simple implementation L_{spinlock} of a spinlock on TSO. We consider only well-behaved clients of the spinlock, which, e.g., do not call `release` without having previously called `acquire` (this can be easily taken into account by restricting the most general client appropriately). The `tryacquire` method tries to acquire the lock, but, unlike `acquire`, does not wait for it to be released if it is busy; it just returns 0 in this case. For efficiency, `release` writes 1 to `x` without executing a memory barrier. This optimisation is used, e.g., by implementations of spinlocks in the Linux kernel [5]. On TSO this can result in an additional delay before the write releasing the lock becomes visible to another CPU trying to acquire it. As a consequence, `tryacquire` can return 0 even after the lock has actually been released. For example, the following is a valid history of the spinlock implementation on TSO, which cannot be produced on an SC memory model:

$$(1, \mathsf{call}\ \mathtt{acquire})\ (1, \mathsf{ret}\ \mathtt{acquire})\ (1, \mathsf{call}\ \mathtt{release})\ (1, \mathsf{ret}\ \mathtt{release})$$
$$(2, \mathsf{call}\ \mathtt{tryacquire})\ (2, \mathsf{ret}\ \mathtt{tryacquire}(0)). \quad (1)$$

Figure 1b shows an abstract SC implementation $L^{\sharp}_{\mathsf{spinlock}}$ of the spinlock capturing the behaviours of its concrete TSO implementation, such as the one given by the above history. Here `release` writes 1 to `x` immediately. To capture the effects of the concrete library implementation running on TSO, the SC specification is weaker than might be expected: `tryacquire` in Figure 1b can spuriously return 0 even when `x` contains 1.

PROPOSITION 4. $L_{\mathsf{spinlock}} \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^{\sharp}_{\mathsf{spinlock}}$.

The same specification is also suitable for more complicated spinlock implementations [7, Appendix B]. We note that the weak specification of `tryacquire` has been adopted by the C++ memory model [3] to allow certain compiler optimisations. As we show in Section 5, linearizability with respect to an SC specification ensures the correctness of implementations of `tryacquire` and other synchronisation primitives comprising the run-time system of a DRF language. Our example thus shows that the specification used in C++ is also needed to capture the behaviour of common spinlock implementations.

Correctness of TSO-to-SC Linearizability. A good notion of library specification has to allow replacing a library implementation with its specification in reasoning about a client. We now show that the notion of TSO-to-SC linearizability proposed above satisfies a variant of this property. To reason about clients of TSO libraries with respect to SC specifications of the latter, we consider a mixed ***TSO/SC semantics*** of programs, which executes the client on TSO and the library on SC. That is, read and write commands by the library code bypass the store buffer and access the memory directly (the formal semantics is given in [7, Appendix B]). We denote the set of traces of a program $C(L)$ in this semantics with $[\![C(L)]\!]_{\mathsf{TSO/SC}}$.

To express properties of a client preserved by replacing the implementation of the library it uses with its specification, we introduce the following operation. For a trace τ of $C(L)$, let $\mathsf{client}(\tau)$ be its projection to actions relevant to the client, i.e., executed by the client code or corresponding to flushes of client entries in store buffers. Formally, we include an action $\varphi = (t, \_)$ such that $\tau = \tau'\varphi\tau''$ into the projection if:

- φ is an interface action, i.e., a call or a return; or
- φ is not a flush or an interface action, and it is not the case that $\tau|_t = \tau_1\,(t, \mathsf{call}\ \_)\,\tau_2\varphi\tau_3$, where τ_2 does not contain a $(t, \mathsf{ret}\ \_)$ action; or
- $\varphi = (\_, \mathsf{flush}(x, \_))$ for some $x \in \mathsf{CLoc}$.

We lift client to sets T of traces pointwise: $\mathsf{client}(T) = \{\mathsf{client}(\tau) \mid \tau \in T\}$.

THEOREM 5 (Abstraction to SC). *If* $L_1 \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L_2$, *then* $\mathsf{client}([\![C(L_1)]\!]_{\mathsf{TSO}}) \subseteq \mathsf{client}([\![C(L_2)]\!]_{\mathsf{TSO/SC}})$.

According to Theorem 5, while reasoning about a client $C(L_1)$ of a TSO library L_1, we can soundly replace L_1 with its SC version L_2 linearizing L_1: if a trace property over client actions holds of $C(L_2)$, it will also hold of $C(L_1)$. The theorem can thus be used to simplify reasoning about TSO programs. Although Theorem 5 is not the main contribution of this paper, it serves as a sanity check for our definition of linearizability, and is useful for discussing our main technical result in Section 5.

Compositionality of TSO-to-SC Linearizability. The following corollary of Theorem 5 states that, like the classical notion of linearizability [9], ours is compositional: if several non-interacting libraries are linearizable, so is their composition. This allows extending the results presented in the rest of the paper to programs with multiple libraries. Formally, consider libraries $L_1, \ldots, L_k$ with disjoint sets of declared methods and assume that the set of library locations LLoc is partitioned into locations belonging to every library: $\mathsf{LLoc} = \mathsf{LLoc}_1 \uplus \ldots \uplus \mathsf{LLoc}_k$. We assume that, in any program, a library L_j accesses only locations from LLoc_j. We let L, respectively, $L^\sharp$ be the library implementing all of the methods from $L_1, \ldots, L_k$, respectively, $L_1^\sharp, \ldots, L_k^\sharp$.

COROLLARY 6 (Compositionality). *If* $\forall j.\, L_j \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L_j^\sharp$*, then* $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^\sharp$.

Comparison with TSO-to-TSO Linearizability. As the abstract library implementation in TSO-to-SC linearizability executes on SC, it does not describe how the concrete library implementation uses store buffers. TSO libraries can also be specified by abstract implementations running on TSO, which do describe this usage. In [6], we proposed the notion of TSO-to-TSO linearizability $\sqsubseteq_{\mathsf{TSO}\to\mathsf{TSO}}$ between two TSO libraries, which validates the following version of the Abstraction Theorem.

THEOREM 7 (Abstraction to TSO). *If* $L_1 \sqsubseteq_{\mathsf{TSO}\to\mathsf{TSO}} L_2$*, then* $\mathsf{client}(\llbracket C(L_1)\rrbracket_{\mathsf{TSO}}) \subseteq \mathsf{client}(\llbracket C(L_2)\rrbracket_{\mathsf{TSO}})$.

The particularities of TSO-to-TSO linearizability are not relevant here; suffice it to say that the definition requires that the two libraries use store buffers in similar ways, and to this end, enriches histories with extra actions. The spinlock from Figure 1a has the abstract TSO implementation with `acquire` and `release` implemented as in Figure 1b, and `tryacquire`, as in Figure 1a (the implementation and the specification of `tryacquire` are identical in this case because the spinlock considered is very simple; see [7, Appendix B] for more complicated cases). Since the specification executes on TSO, the write to `x` in `release` can be delayed in the store buffer. In exchange, the specification of `tryacquire` does not include spurious failures.

Both TSO-to-SC and TSO-to-TSO linearizability validate versions of the Abstraction Theorem (Theorems 5 and 7). The theorem validated by TSO-to-SC is weaker than the one validated by TSO-to-TSO: a property of a client of a library may be provable after replacing the latter with its TSO specification using Theorem 7, but not after replacing it with its SC specification using Theorem 5. Indeed, consider the following client of the spinlock in Figure 1a, where `a` and `b` are local to the second thread:

```
                          u = 0;
acquire(); release(); u = 1;  ||  a = u; b = tryacquire();
```

$$\{\mathtt{a} = 1 \Rightarrow \mathtt{b} = 1\}$$

The postcondition shown holds of the program: since store buffers in TSO are FIFO, if the write to `u` has been flushed, so has been the write to `x` in `release`, and `tryacquire` has to succeed. However, it cannot be established after we apply Theorem 5 with the spinlock specification in Figure 1b, as the abstract implementation of `tryacquire` returns an arbitrary result when the lock is free. The postcondition can still be established after we apply Theorem 7 with the TSO specification of the spinlock given in Section 3, since the specification allows us to reason about the correlations in the use of store buffers by the library and the client. To summarise, SC specifications of TSO libraries trade the weakness of the memory model for the weakness of the specification.

Example: Seqlock. We now consider an example of a TSO library whose SC specification is more subtle than that of a spinlock. Figure 2 presents a simplified version L_{seqlock}

```
word x1 = 0, x2 = 0, c = 0;             read(out word d1, out word d2) {
                                          word c0;
write(in word d1, in word d2) {           do {
  c++;                                      do { c0 = c; } while (c0 % 2);
  x1 = d1; x2 = d2;                         d1 = x1; d2 = x2;
  c++;                                    } while (c != c0);
}                                       }
```

Fig. 2. L_{seqlock}: a TSO seqlock implementation

of a seqlock [5]—an efficient implementation of a readers-writer protocol based on version counters used in the Linux kernel. Two memory addresses `x1` and `x2` make up a conceptual register that a single hardware thread can write to, and any number of other threads can read from. A version number is stored at `c`. The writing thread maintains the invariant that the version is odd during writing by incrementing it before the start of and after the finish of writing. A reader checks that the version number is even before attempting to read. After reading, it checks that the version has not changed, thereby ensuring that no write has overlapped the read. Neither `write` nor `read` includes a barrier, so that writes to `x1`, `x2` and `c` may not be visible to readers immediately.

An SC specification for the seqlock is better given not by the source code of an abstract implementation, like in the case of a spinlock, but by explicitly describing the set of its histories $\mathsf{history}([\![L_2]\!]_{\mathsf{SC}})$ to be used in Definition 2 (an operational specification also exists, but is more complicated; see [7, Appendix B]). We now adjust the definition of TSO-to-SC linearizability to accept a library specification defined in this way.

Specifying Libraries by Sets of Histories. For a TSO library L and a set of histories T, we let $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} T$, if $\forall H_1 \in \mathsf{history}([\![L]\!]_{\mathsf{TSO}}).\, \exists H_2 \in T.\, H_1 \sqsubseteq H_2$. The formulation of Theorem 5 can be easily adjusted to accommodate this notion of linearizability.

We now give a specification to the seqlock as a set of histories T_{seqlock}. First of all, methods of a seqlock should appear to take effect atomically. Thus, in histories from T_{seqlock}, if call action has a matching return, then the latter has to follow it immediately. Consider a history H_0 satisfying this property. Let $\mathsf{writes}(H_0)$ be the sequence of pairs (d_1, d_2) from actions of the form $(\_, \mathsf{call}\ \mathtt{write}(d_1, d_2))$ in H_0, and $\mathsf{reads}(H_0)$, the sequence of (d_1, d_2) from actions of the form $(\_, \mathsf{ret}\ \mathtt{read}(d_1, d_2))$. For a sequence α, let $\alpha^\dagger$ be its stutter-closure, i.e., the set of sequences obtained from α by repeating some of its elements. We lift the stutter-closure operation to sets of sequences pointwise. Given the above definitions, a history H belongs to T_{seqlock} if for every prefix H_0 of H, $\mathsf{reads}(H_0)$ is a subsequence of a sequence from $((0, 0)\, \mathsf{writes}(H_0))^\dagger$. Recall that a seqlock allows only a single thread to call `write`. This specification thus ensures that readers see the writes in the order it issues them, but possibly with a delay.

PROPOSITION 8. $L_{\mathsf{seqlock}} \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} T_{\mathsf{seqlock}}$.

4 TSO-to-SC Linearizability and Robustness

One way to simplify reasoning about a TSO program is by checking that it is ***robust***, meaning that it produces only those externally visible behaviours that could also be obtained by running it on an SC machine. Its properties can then be proved by considering only its SC executions. Several criteria for checking robustness of TSO programs have been proposed recently [12,4,1]. TSO-to-SC linearizability is more flexible than such criteria: since an abstract library implementation can have different source code than its concrete implementation, it allows the latter to have non-SC behaviours. However, checking the requirements of a robustness criterion is usually easier than proving linearizability. We therefore show how one such criterion, data-race freedom, can be used to simplify establishing TSO-to-SC linearizability when it is applicable. On the way, we introduce some of the technical ingredients necessary for our main result in Section 5.

We first define the notion of DRF for the low-level machine of Section 2. Our intention is that the DRF of a program must ensure that it produces only SC behaviours (see Theorem 10 below). All robustness criteria proposed so far have assumed a closed program P consisting of a client that does not use a library. We define the robustness of libraries using the most general client of Section 3. For a trace fragment τ with all actions by a thread t, we denote with $\mathsf{block}(\tau)$ a trace of one of the following two forms: τ or $(t, \mathsf{lock})\, \tau_1\, \tau\, \tau_2\, (t, \mathsf{unlock})$, where τ_1, τ_2 do not contain (t, unlock).

DEFINITION 9. *A **data race** is a fragment of an SC trace of the form* $\mathsf{block}(\tau)\, (t', \mathsf{write}(x, \_))\, (t', \mathsf{flush}(x, \_))$*, where* $\tau \in \{(t, \mathsf{write}(x, \_))\, (t, \mathsf{flush}(x, \_)), (t, \mathsf{read}(x, \_))\}$ *and* $t \neq t'$*. A program P is **data-race free (DRF)**, if so are traces in* $[\![P]\!]_{\mathsf{SC}}$*; a library L is DRF, if so are traces in* $[\![L]\!]_{\mathsf{SC}}$.

Thus, a race is a memory access followed by a write to the same location, where the former, but not the latter, can be in a lock..unlock block. This is a standard notion of a data race with one difference: even though $(t, \mathsf{read}(x))\, (t', \mathsf{write}(x))\, (t', \mathsf{flush}(x))$ is a race, $(t, \mathsf{read}(x))\, (t', \mathsf{lock})\, (t', \mathsf{write}(x))\, (t', \mathsf{flush}(x))\, (t', \mathsf{unlock})$ is not. We do not consider conflicting accesses of the latter kind (e.g., with the write inside a CAS, which includes a memory barrier) as a race, since they do not lead to a non-SC behaviour.

We adopt the following formalisation of externally visible program behaviours. Assume a set $\mathsf{VLoc} \subseteq \mathsf{CLoc}$ of client locations whose values in memory can be observed during a program execution by its environment. ***Visible actions*** are those of the form $(t, \mathsf{read}(x, u))$ or $(t, \mathsf{flush}(x, u))$, where $x \in \mathsf{VLoc}$. We let $\mathsf{visible}(\tau)$ be the projection of τ to visible actions, and lift visible to sets T of traces pointwise: $\mathsf{visible}(T) = \{\mathsf{visible}(\tau) \mid \tau \in T\}$. Visible locations are ***protected*** in $C(L)$, if every visible action in a trace from $[\![C(L)]\!]_{\mathsf{TSO}}$ occurs within a lock..unlock block. On x86, this requires a memory barrier after every output action, thus ensuring that it becomes visible immediately. The following is a folklore robustness result.

THEOREM 10 (Robustness via DRF). *If P is DRF and visible locations are protected in it, then* $\mathsf{visible}([\![P]\!]_{\mathsf{TSO}}) \subseteq \mathsf{visible}([\![P]\!]_{\mathsf{SC}})$.

Note that here DRF is checked on the SC semantics and at the same time implies that the program behaves SC. This circularity is crucial for using results such as Theorem 10 to simplify reasoning, as it allows not considering TSO executions at all.

THEOREM 11 (Linearizability via DRF). *If L is DRF, then* $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L$.

This allows using classical linearizability [9] to establish TSO-to-SC one by linearizing the library L running on SC to its SC specification $L^\sharp$, thus yielding $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^\sharp$. This can then be used for modular reasoning by applying Theorem 5.

Many concurrent algorithms on TSO (e.g., the classical Treiber's stack) are DRF, as they modify the data structure using only CAS operations, which include a memory barrier. Hence, their linearizability with respect to SC specifications can be established using Theorem 11. However, the DRF criterion may sometimes be too strong: e.g., in the spinlock implementation from Figure 1a, the read from `x` in `acquire` and the write to it in `release` race. We consider more flexible robustness criteria in Section 6.

5 Conditions for Correct Compilation

Our goal in this section is to extend DRF memory models of high-level languages to the case of programs using native TSO libraries, and to identify conditions under which the compiler implements the models correctly. We start by presenting the main technical result of the paper that enables this—the Simulation Theorem.

Simulation Theorem. Consider a program C, meant to be compiled from a high-level language with a DRF model, which uses a native TSO library L. We wish to determine the conditions under which the program produces only SC behaviours, despite possible races inside L. To this end, we first generalise DRF on TSO (Definition 9) to such programs. We define DRF with respect to an SC specification $L^\sharp$ of L.

DEFINITION 12. *$C(L^\sharp)$ is **DRF** if so is any trace from* $\mathsf{client}([\![C(L^\sharp)]\!]_{\mathsf{SC}})$.

This allows races inside the library code, as its internal behaviour is of no concern to the client. Note that checking DRF does not require reasoning about weak memory.

THEOREM 13 (Simulation). *If $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^\sharp$, $C(L^\sharp)$ is DRF, and visible locations are protected in $C(L)$, then* $\mathsf{visible}([\![C(L)]\!]_{\mathsf{TSO}}) \subseteq \mathsf{visible}([\![C(L^\sharp)]\!]_{\mathsf{SC}})$.

Thus, the behaviour of a DRF client of a TSO library can be reproduced when the client executes on the SC memory model and uses a TSO-to-SC linearization of the library implementation. Note that the DRF of the client is defined with respect to the SC specification $L^\sharp$ of the TSO library L. Replacing L by $L^\sharp$ allows hiding non-SC behaviours internal to the library, which are of no concern to the client. Corollary 6 allows applying the theorem to clients using multiple libraries.

Extending Memory Models of High-Level Languages. We describe a method for extending a high-level memory model to programs with native TSO libraries in general terms, without tying ourselves to its formalisation. We give an instantiation for the case of the C++ memory model (excluding weak atomics) in [7, Appendix B]. Consider a high-level language with a DRF memory model. That is, we assume an SC semantics for the language, and a notion of DRF on this semantics. For a program $\mathcal{P}$ in this language, let $[\![\mathcal{P}]\!]$ be the set of its externally visible behaviours resulting from its executions in the semantics of the high-level language. At this point, we do not need to define what these behaviours are; they might include, e.g., input/output information.

Let $\mathcal{C}(L)$ be a program in a high-level language using a TSO library L with an SC specification $L^\sharp$, i.e., $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^\sharp$. The specification $L^\sharp$ allows us to extend the semantics of the language to describe the intended behaviour of $\mathcal{C}(L)$. Informally, we let the semantics of calling a method of L be the effect of the corresponding method of $L^\sharp$. As both $\mathcal{C}$ and $L^\sharp$ are meant to have an SC semantics, the effect of $L^\sharp$ can be described within the memory model of the high-level language.

To define this extension more formally, it is convenient for us to use the specification of $L^\sharp$ given by its set of histories $\mathsf{history}([\![L^\sharp]\!]_{\mathsf{SC}})$, rather than by its source code, as this sidesteps the issues arising when composing the sources of programs in a low-level language and a high-level one. Namely, we define the semantics of $\mathcal{C}(L)$ in two stages. First, we consider the set of executions of $\mathcal{C}(L)$ in the semantics of the high-level language where a call to a method of L is interpreted in the same way as a call to a method of the high-level language returning arbitrary values. Since the high-level language has an SC semantics, every program execution in it is a trace obtained by interleaving actions of different threads, which has a single history of calls to and returns from L. We then define the intended behaviour of $\mathcal{C}(L)$ by the set $[\![\mathcal{C}(L^\sharp)]\!]$ of externally visible behaviours resulting from the executions that have a history from $\mathsf{history}([\![L^\sharp]\!]_{\mathsf{SC}})$[2]. This semantics also generalises the notion of DRF to the extended language: programs are DRF when the executions of $\mathcal{C}(L)$ selected above have no races between client actions as defined for high-level programs without TSO libraries. In particular, DRF is defined with respect to SC specifications of libraries that the client uses, not their TSO implementations.

From the point of view of the extended memory model, the run-time system of the high-level language, implementing built-in synchronisation primitives, is no different from external TSO libraries. The extension thus allows deriving a memory model consistent with the implementation of synchronisation primitives on TSO (e.g., spinlocks or seqlocks from Section 3) from the memory model of the base language excluding the primitives. Below, we use this fact to separate the reasoning about the correctness of a compiler for the high-level language from that about the correctness of its run-time system. This approach to deriving the memory model does not result in imprecise specifications: e.g., the SC specification of a TSO spinlock implementation in Section 3 corresponds to the one in the C++ standard.

Conditions for Correct Compilation. Theorem 13 allows us to formulate conditions under which a compiler from a high-level DRF language correctly implements the extended memory model defined above. Let $\langle\mathcal{C}\rangle(L)$ be the compilation of a program $\mathcal{C}$ in the high-level language to the TSO machine from Section 2, linked with a native TSO library L. Assume an SC specification $L^\sharp$ of L:

(i) $L \sqsubseteq_{\mathsf{TSO}\to\mathsf{SC}} L^\sharp$.

Then the extended memory model defines the intended semantics $[\![\mathcal{C}(L^\sharp)]\!]$ of the program. Let us denote the compiled code linked with $L^\sharp$, instead of L, as $\langle\mathcal{C}\rangle(L^\sharp)$. We place the following constraints on the compiler:

(ii) $\mathcal{C}$ is correctly compiled to an SC machine: $\mathsf{visible}([\![\langle\mathcal{C}\rangle(L^\sharp)]\!]_{\mathsf{SC}}) \subseteq [\![\mathcal{C}(L^\sharp)]\!]$.

(iii) $\langle\mathcal{C}\rangle(L^\sharp)$ is DRF, i.e., so are all traces from $\mathsf{client}([\![\langle\mathcal{C}\rangle(L^\sharp)]\!]_{\mathsf{SC}})$.

(iv) Visible locations are protected in $\langle\mathcal{C}\rangle(L)$.

[2] Here we assume that language-level threads correspond directly to hardware-level ones. This assumption is sound even when the actual language implementation multiplexes several threads onto fewer CPUs using a scheduler, provided the latter executes a memory barrier at every context switch; see [7, Appendix B] for discussion.

From Theorem 13 and (i), (iii) and (iv), we obtain $\mathsf{visible}([\![\langle\mathcal{C}\rangle(L)]\!]_{\mathsf{TSO}}) \subseteq \mathsf{visible}([\![\langle\mathcal{C}\rangle(L^\sharp)]\!]_{\mathsf{SC}})$, which, together with (ii), implies $\mathsf{visible}([\![\langle\mathcal{C}\rangle(L)]\!]_{\mathsf{TSO}}) \subseteq [\![\mathcal{C}(L^\sharp)]\!]$. Hence, any observable behaviour of the compiled code using the TSO library implementation is included into the intended semantics of the program defined by the extended memory model. Therefore, our conditions entail the compiler correctness.

The conditions allow for a separate consideration of the hardware memory model and the run-time system implementation when reasoning about the correctness of a compiler from a DRF language to a TSO machine. Namely, (ii) checks the correctness of the compiler while ignoring the fact that the target machine has a weak memory model and assuming that the run-time system is implemented correctly. Conditions (iii) and (iv) then ensure the correctness of the compiled code on TSO, and condition (i), the correctness of the run-time system.

Establishing (iii) requires ensuring the DRF of the compiled code given the DRF of the source program in the high-level language. In practice, this might require the compiler to insert additional memory barriers. For example, the SC fragment of C++ [3,2] includes so-called ***strong atomic*** operations, whose concurrent accesses to the same location are not considered a race. The DRF of the high-level program thus ensures that, in the compiled code, we cannot have a race in the sense of Definition 9, except between instructions resulting from strong atomic operations. To prevent the latter, existing barrier placement schemes for C++ compilation on TSO [2] include a memory barrier when translating a strong atomic write. As this prevents a race in the sense of Definition 9, these compilation schemes satisfy our conditions.

Discussion. Theorem 13 is more subtle than might seem at first sight. The crux of the matter is that, like Theorem 10, it allows checking DRF on the SC semantics of the program. This makes the theorem powerful in practice, but requires its proof to show that a trace from $[\![C(L)]\!]_{\mathsf{TSO}}$ with a visible non-SC behaviour can be converted into one from $[\![C(L^\sharp)]\!]_{\mathsf{SC}}$ exhibiting a race. Proving this is non-trivial. A naive attempt to prove the theorem might first replace L with its linearization $L^\sharp$ using Theorem 5 and then try to apply a variant of Theorem 10 to show that the resulting program is SC:

$$\mathsf{visible}([\![C(L)]\!]_{\mathsf{TSO}}) \subseteq \mathsf{visible}([\![C(L^\sharp)]\!]_{\mathsf{TSO/SC}}) \subseteq \mathsf{visible}([\![C(L^\sharp)]\!]_{\mathsf{SC}}).$$

However, the second inclusion does not hold even if $C(L^\sharp)$ is DRF, as Theorem 10 does not generalise to the TSO/SC semantics. Indeed, take the spinlock implementation and specification from Figure 1 as L and $L^\sharp$ and consider the following client C:

```
                         x = y = 0;
acquire(); x = 1; release(); || lock; y = 1; unlock;
b = y;                       || acquire(); a = x; release();
                        {a = b = 0}
```

The outcome shown is allowed by $[\![C(L^\sharp)]\!]_{\mathsf{TSO/SC}}$, but disallowed by $[\![C(L^\sharp)]\!]_{\mathsf{SC}}$, even though the latter is DRF. It is also disallowed by $[\![C(L)]\!]_{\mathsf{TSO}}$: in this case, the first thread can only read 0 from `y` if the second thread has not yet executed `y = 1`; but when the second thread later acquires the lock, the write of 1 to `x` by the first thread is guaranteed to have been flushed into the memory, and so the second thread has to read 1 from x. The trouble is that $[\![C(L^\sharp)]\!]_{\mathsf{TSO/SC}}$ loses such correlations between the store buffer usage by the client and the library, which are important for mapping a non-SC trace from $[\![C(L)]\!]_{\mathsf{TSO}}$ into a racy trace from $[\![C(L^\sharp)]\!]_{\mathsf{SC}}$. The need for maintaining the correlations leads to a subtle proof that uses a non-standard variant of TSO to first make the client part of the trace SC and only then replace the library L with its SC specification $L^\sharp$. See [7, Appendix C] for a more detailed discussion.

6 Using Robustness Criteria More Flexible Than DRF

Assume that code inside a lock..unlock block accesses at most one memory location.

DEFINITION 14. *A **quadrangular race** is a fragment of an SC trace of the form:*

$$(t, \mathsf{write}(x, \_))\, \tau_1\, (t, \mathsf{read}(y, \_))\, \mathsf{block}((t', \mathsf{write}(y, \_))\, (t', \mathsf{flush}(y, \_)))\, \tau_2\, \mathsf{block}(\varphi),$$

where $\varphi \in \{(t'', \mathsf{write}(x, \_))\, (t'', \mathsf{flush}(x, \_)), (t'', \mathsf{read}(x, \_))\}$*,* $t \neq t'$*,* $t \neq t''$*,* $x \neq y$*,* τ_1 *contains only actions by* t*, and* τ_1, τ_2 *do not contain* (t, unlock)*. A program* P *is **quadrangular-race free (QRF)**, if so are traces in* $[\![P]\!]_{\mathsf{SC}}$*.*

THEOREM 15 (Robustness via QRF). *If* P *is QRF and visible locations are protected in it, then* $\mathsf{visible}([\![P]\!]_{\mathsf{TSO}}) \subseteq \mathsf{visible}([\![P]\!]_{\mathsf{SC}})$.

This improves on a criterion by Owens [12], which does not require the last access to x, and thus falsely signals a possible non-SC behaviour when x is local to thread t.

Unfortunately, QRF cannot be used to simplify establishing TSO-to-SC linearizability, because Theorem 11 does not hold if we assume only that L is QRF. Intuitively, transforming a TSO trace satisfying QRF into an SC one can rearrange calls and returns in ways that break linearizability. Formally, the spinlock L_{spinlock} in Figure 1a is QRF, and its history (1) has a single linearization—itself. However, it cannot be reproduced when executing L_{spinlock} on an SC memory model. Moreover, the QRF of a library does not imply Theorem 13 for $L^\sharp = L$ [7, Appendix B]. We now show that Theorem 13 can be recovered for QRF libraries under a stronger assumption on the client.

DEFINITION 16. *A program* $C(L)$ *is **strongly DRF** if it is DRF and traces in* $\mathsf{client}([\![C(L)]\!]_{\mathsf{SC}})$ *do not contain fragments of the form* $(t, \mathsf{read}(x, \_))\, \mathsf{block}((t', \mathsf{write}(x, \_))\, (t', \mathsf{flush}(x, \_)))$*, where* $t \neq t'$*.*

THEOREM 17. *If* L *is QRF,* $C(L)$ *is strongly DRF, and visible locations are protected in it, then* $\mathsf{visible}([\![C(L)]\!]_{\mathsf{TSO}}) \subseteq \mathsf{visible}([\![C(L)]\!]_{\mathsf{SC}})$.

When C is compiled from C++, the requirement that C be strongly DRF prohibits the C++ program from using strong atomic operations, which is restrictive.

7 Related Work

To the best of our knowledge, there has been no research on modularly checking the interoperability between components written for different language and hardware memory models. For example, the existing proof of correctness of C++ compilation to x86 [2] does not consider the possibility of a C++ program using arbitrary native components and assume fixed implementations of C++ synchronisation primitives in the run-time system. In particular, the correctness proofs would no longer be valid if we changed the run-time system implementation. As we discuss in Section 5, this paper provides conditions for an *arbitrary* run-time system implementation of a DRF language ensuring the correctness of the compilation.

We have previously proposed a generalisation of linearizability to the TSO memory model [6] (TSO-to-TSO linearizability in Section 3). Unlike TSO-to-SC linearizability, it requires specifications to be formulated in terms of low-level hardware concepts, and thus cannot be used for interfacing with high-level languages. Furthermore, the technical focus of [6] was on establishing Theorem 7, not Theorem 13.

To concentrate on the core issues of handling interoperability between TSO and DRF models, we assumed that the data structures of the client and its libraries are completely disjoint. Recently, we have proposed a generalisation of classical linearizability that allows the client to communicate with the libraries via data structures [8]. We hope that the results from the two papers can be combined to lift the above restriction.

Acknowledgements. We thank Matthew Parkinson and Serdar Tasiran for comments that helped to improve the paper. Yang was supported by EPSRC.

References

1. Alglave, J., Maranget, L.: Stability in Weak Memory Models. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 50–66. Springer, Heidelberg (2011)
2. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL (2011)
3. Boehm, H.-J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: PLDI (2008)
4. Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding Robustness against Total Store Ordering. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 428–440. Springer, Heidelberg (2011)
5. Bovet, D., Cesati, M.: Understanding the Linux Kernel, 3rd edn. O'Reilly (2005)
6. Burckhardt, S., Gotsman, A., Musuvathi, M., Yang, H.: Concurrent Library Correctness on the TSO Memory Model. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 87–107. Springer, Heidelberg (2012)
7. Gotsman, A., Musuvathi, M., Yang, H.: Show no weakness: Sequentially consistent specifications of TSO libraries (extended version) (2012), `http://www.software.imdea.org/~gotsman`
8. Gotsman, A., Yang, H.: Linearizability with Ownership Transfer. In: Ulidowski, I. (ed.) CONCUR 2012. LNCS, vol. 7454, pp. 256–271. Springer, Heidelberg (2012)
9. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. In: TOPLAS (1990)
10. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comp. (1979)
11. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL (2005)
12. Owens, S.: Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010)
13. Owens, S., Sarkar, S., Sewell, P.: A Better x86 Memory Model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009)

Collecting Information by Power-Aware Mobile Agents[⋆]

Julian Anaya[1], Jérémie Chalopin[2], Jurek Czyzowicz[1], Arnaud Labourel[2], Andrzej Pelc[1], and Yann Vaxès[2]

[1] Université du Québec en Outaouais, C.P. 1250, succ. Hull, Gatineau, Qc. J8X 3X7 Canada
ingjuliananaya@gmail.com, {jurek,pelc}@uqo.ca
[2] LIF, CNRS & Aix-Marseille University, 13453 Marseille, France
{jeremie.chalopin,arnaud.labourel,yann.vaxes}@lif.univ-mrs.fr

Abstract. A set of identical, mobile agents is deployed in a weighted network. Each agent possesses a battery - a power source allowing the agent to move along network edges. Agents use their batteries proportionally to the distance traveled. At the beginning, each agent has its initial information. Agents exchange the actually possessed information when they meet. The agents collaborate in order to perform an efficient *convergecast*, where the initial information of all agents must be eventually transmitted to some agent.

The objective of this paper is to investigate what is the minimal value of power, initially available to all agents, so that convergecast may be achieved. We study the question in the centralized and the distributed settings. In the distributed setting every agent has to perform an algorithm being unaware of the network. We give a linear-time centralized algorithm solving the problem for line networks. We give a 2-competitive distributed algorithm achieving convergecast for tree networks. The competitive ratio of 2 is proved to be the best possible for this problem, even if we only consider line networks. We show that already for the case of tree networks the centralized problem is strongly NP-complete. We give a 2-approximation centralized algorithm for general graphs.

1 Introduction

The Model and the Problem. A set of agents is deployed in a network represented by a weighted graph G. An edge weight represents its length, i.e., the distance between its endpoints along the edge. The agents start at different nodes of G. Every agent has a battery : a power source allowing it to move in a continuous way along the network edges. An agent may stop at any point of a network edge (i.e. at any distance from the edge endpoints, up to the edge weight). The movements of an agent use its battery proportionally to the distance traveled. We assume that all agents move at the same speed that is equal to one, i.e., the values of the distance traveled and the time spent while travelling are equivalent. Each agent starts with the same amount of power noted P, allowing all agents to travel the same distance P.

Initially, each agent has an individual piece of information. When two (or more) agents are at the same point of the network at the same time, they automatically detect

⋆ Partially supported by NSERC discovery grant and by the Research Chair in Distributed Computing at the Université du Québec en Outaouais.

each other's presence and they exchange their information, i.e., each agent transmits all its possessed information to all other agents present at the point (hence an agent transmits information collected during all previous meetings). The purpose of a *convergecast* algorithm is to schedule the movements of the agents, so that the exchanges of the currently possessed information between the meeting agents eventually result in some agent, not a priori predetermined, holding the union of individual information of all the agents. This task is important, e.g., when agents have partial information about the topology of the network and the aggregate information can be used to construct a map of it, or when individual agents hold measurements performed by sensors located at their initial positions and collected information serves to make some global decision based on all measurements.

Agents try to cooperate so that the convergecast is achieved with the agent's smallest possible initial battery power P_{OPT}, i.e., minimizing the maximum distance traveled by an agent. We investigate the problem in two possible settings, centralized and distributed.

In the centralized setting, the problem must be solved by a centralized authority knowing the network and the initial positions of all the agents. We define a *strategy* as a finite sequence of movements executed by the agents. During each movement, starting at a specific time, an agent walks between two points belonging to the same network edge. A strategy is a convergecast strategy if the sequence of movements results in one agent possessing the initial information of every agent. We consider two different versions of the problem : the decision problem, i.e., deciding if there exists a convergecast strategy using power P (where P is the input of the problem) and the optimization problem, i.e., computing the smallest amount of power that is sufficient to achieve convergecast.

In the distributed setting, the problem must be approached individually by each agent. Each agent is unaware of the network, of its position in the network and without the knowledge of positions (or even the presence) of any other agents. The agents are anonymous, i.e., they must execute the same algorithm. The agent has a very simple sensing device allowing it to detect the presence of other agents at its current location in the network. The agent is also aware of the degree d of the node at which it is located and it can identify all ports represented by integers $1, 2, \dots d$. The agent is aware of the directions from which are coming all agents currently present at its location (i.e. their entry ports to the current node or their incoming directions if currently located inside an edge). Each agent has memory sufficient to store all information initially belonging to all agents as well as a small (constant) number of real values. Since the measure of efficiency in this paper is the battery power (or the maximum distance traveled by an agent, which is proportional to the battery power used) we do not try to optimize the other resources (e.g. global execution time, local computation time, memory size of the agents, communication bandwidth, etc.). In particular, we conservatively suppose that, whenever two agents meet, they automatically exchange the entire information they possess (rather than the new information only). This information exchange procedure is never explicitly mentioned in our algorithms, supposing, by default, that it always takes place when a meeting occurs. The efficiency of a distributed solution is expressed by the *competitive ratio*, which is the worst-case ratio of the amount of power necessary to solve the convergecast by the distributed algorithm with respect to the amount of power computed by the optimal centralized algorithm, which is executed for the same agents' initial positions.

It is easy to see, that in the optimal centralized solution for the case of the line and the tree, the original network may be truncated by removing some portions and leaving only the connected part of it containing all the agents (this way all leaves of the remaining tree contain initial positions of agents). We make this assumption also in the distributed setting, since no finite competitive ratio is achievable if this condition is dropped. Indeed, two nearby anonymous agents inside a long line need to travel a long distance to one of its endpoints to break symmetry in order to meet.

Related Work. The rapid development of network and computer industry fueled the research interest in mobile agents (robots) computing. Mobile agents are often interpreted as software agents, i.e., programs migrating from host to host in a network, performing some specific tasks. However, the recent developments in computer technology bring up specific problems related to physical mobile devices. These include robots or motor vehicles, various wireless gadgets, or even living mobile agents: humans (e.g. soldiers on the battlefield or emergency disaster relief personnel) or animals (e.g. birds, swarms of insects).

In many applications the involved mobile agents are small and have to be produced at low cost in massive numbers. Consequently, in many papers, the computational power of mobile agents is assumed to be very limited and feasibility of some important distributed tasks for such collections of agents is investigated. For example [6] introduced *population protocols*, modeling wireless sensor networks by extremely limited finite-state computational devices. The agents of population protocols move according to some mobility pattern totally out of their control and they interact randomly in pairs. This is called *passive mobility*, intended to model, e.g., some unstable environment, like a flow of water, chemical solution, human blood, wind or unpredictable mobility of agents' carriers (e.g. vehicles or flocks of birds). On the other hand, [37] introduced anonymous, oblivious, asynchronous, mobile agents which cannot directly communicate, but can occasionally observe the environment. Gathering and convergence [5,19,20,21], as well as pattern formation [23,25,37,38] were studied for such agents.

Apart from the feasibility questions for such limited agents, the optimization problems related to the efficient usage of agents' resources have been also investigated. Energy management of (not necessarily mobile) computational devices has been a major concern in recent research papers (cf. [1]). Fundamental techniques proposed to reduce power consumption of computer systems include power-down strategies (see [1,8,30]) and speed scaling (introduced in [39]). Several papers proposed centralized [17,36,39] or distributed [1,4,8,30] algorithms. However, most of this research on power efficiency concerned optimization of overall power used. Similar to our setting, assignment of charges to the system components in order to minimize the maximal charge has a flavor of another important optimization problem which is load balancing (cf. [10]).

In wireless sensor and ad hoc networks the power awareness has been often related to the data communication via efficient routing protocols (e.g. [4,36]. However in many applications of mobile agents (e.g. those involving actively mobile, physical agents) the agent's energy is mostly used for its mobility purpose rather than communication, since active moving often requires running some mechanical components, while communication mostly involves (less energy-prone) electronic devices. Consequently, in most tasks involving moving agents, like exploration, searching or pattern formation, the distance traveled is the main optimization criterion (cf. [2,3,11,12,15,16,22,24,26,33]). Single agent exploration of an unknown environment has been studied for graphs, e.g. [2,22], or geometric terrains, [12,16].

While a single agent cannot explore an unknown graph unless pebble (landmark) usage is permitted (see [13]), a pair of robots is able to explore and map a directed graph of maximal degree d in $O(d^2 n^5)$ time with high probability (cf. [14]). In the case of a team of collaborating mobile agents, the challenge is to balance the workload among the agents so that the time to achieve the required goal is minimized. However this task is often hard (cf. [28]), even in the case of two agents on a tree, [9]. On the other hand, [26] study the problem of agents exploring a tree showing $O(k/\log k)$ competitive ratio of their distributed algorithm provided that writing (and reading) at tree nodes is permitted.

Assumptions similar to our paper have been made in [11,16,24] where the mobile agents are constrained to travel a fixed distance to explore an unknown graph, [11,16], or tree, [24]. In [11,16] a mobile agent has to return to its home base to refuel (or recharge its battery) so that the same maximal distance may repeatedly be traversed. [24] gives an 8-competitive distributed algorithm for a set of agents with the same amount of power exploring the tree starting at the same node.

The convergecast problem is sometimes viewed as a special case of the data aggregation question (e.g. [32,35]) and it has been studied mainly for wireless and sensor networks, where the battery power usage is an important issue (cf. [31,7]). Recently [18] considered the online and offline settings of the scheduling problem when data has to be delivered to mobile clients while they travel within the communication range of wireless stations. [31] presents a randomized distributed convergecast algorithm for geometric ad-hoc networks and studies the trade-off between the energy used and the latency of convergecast. To the best of our knowledge, the problem of the present paper, when the mobile agents perform convergecast, by exchanging the previously acquired information when meeting, while optimizing the maximal power used by a mobile agent, has never been investigated.

Our Results. In the case of centralized setting we give a linear-time deterministic algorithm finding an optimal convergecast strategy for line networks. We show that, already for the case of tree networks, the centralized problem is strongly NP-complete. We give a 2-approximation centralized algorithm for general graphs.

For the distributed setting, we show that the convergecast is possible for tree networks if all agents have the amount of initial power equal to twice the power necessary to achieve centralized convergecast. The competitive ratio of 2 is proved to be the best possible for this problem, even if we only consider line networks. Most proofs are omitted due to lack of space. They will appear in a journal version of this paper.

2 Centralized Convergecast on Lines

In this section we consider the centralized convergecast problem for lines. We give an optimal, linear-time, deterministic centralized algorithm, computing the optimal amount of power needed to solve convergecast for line networks. As the algorithm is quite involved, we start by observing some properties of the optimal strategies. Already relatively apparent properties permit us to design an intuitive decision procedure, verifying whether a given amount of power is sufficient to perform convergecast. Then we present other ingredients needed for the linear-time optimization procedure.

We order agents according to their positions on the line. Hence we can assume w.l.o.g., that agent a_i, for $1 \leq i \leq n$ is initially positioned at point $Pos[i]$ of the line of length ℓ and that $Pos[1] = 0 < Pos[2] < \ldots < Pos[n] = \ell$.

2.1 Properties of a Convergecast Strategy

In this subsection, we show that if we are given a convergecast strategy for some configuration, then we can always modify it in order to get another convergecast strategy, using the same amount of maximal power for every agent, satisfying some interesting properties. These observations permit us to restrict the search for the optimal strategy to some smaller and easier to handle subclass of strategies.

Observe that, in order to aggregate the entire information at a single point of the line, every agent a_i, for $1 < i < n$, must learn either the initial information of agent a_1 or a_n. Therefore, we can partition the set of agents performing a convergecast strategy into two subsets LR and RL, such that each agent $a_i \in LR$ learns the initial information of agent a_1 before learning the initial information of agent a_n (or not learning at all the information of a_n). All other agents belong to RL. For any convergecast strategy all the points visited by agent a_i form a real interval containing its initial position $Pos[i]$. We denote by $[b_i, f_i]$ the interval of all points visited by $a_i \in LR$ and by $[f_j, b_j]$ - the points visited by $a_j \in RL$.

In the next lemma, we show a necessary and sufficient condition for the existence of a convergecast strategy. It also shows that any convergecast strategy may be converted to a strategy that we call *regular* having particular properties. Firstly, each agent from LR of a regular strategy is initially positioned left to all agents of RL. Secondly, each agent of regular strategy needs to change its direction at most once. More precisely, each agent $a_i \in LR$ first goes back to a point $b_i \leq Pos[i]$, getting there the information from the previous agent (except a_1 that has no information to collect), then it goes forward to a point $f_i \geq b_i$. Similarly, each agent in RL first goes back to a point $b_i \geq Pos[i]$ and then moves forward to a point $f_i \leq b_i$. Moreover, we assume that each agent of a regular strategy travels the maximal possible distance, i.e., it spends all its power.

Lemma 1. *There exists a convergecast strategy $\mathcal{S}$ for a configuration $Pos[1:n]$ if and only if there exists a partition of the agents into two sets LR and RL and if for each agent a_i, there exist two points b_i, f_i of segment $[0, \ell]$ such that*

1. *there exists p such that $LR = \{a_i \mid i \leq p\}$ and $RL = \{a_i \mid i > p\}$,*
2. *if $a_i \in LR$, $b_i = \min\{f_{i-1}, Pos[i]\}$ ($b_1 = Pos[1] = 0$) and $f_i = \min\{2b_i + P - Pos[i], \ell\}$,*

3. *if* $a_i \in RL$, $b_i = \max\{f_{i+1}, Pos[i]\}$ *(*$b_n = Pos[n] = \ell$*) and* $f_i = \max\{2b_i - P - Pos[i], 0\}$,
4. $\max\{f_i \mid a_i \in LR\} \geq \min\{f_i \mid a_i \in RL\}$.

In the following, we only consider regular strategies. Note that a regular strategy is fully determined by the value of P and by the partition of the agents into the two sets LR and RL. For each agent $a_i \in LR$ (resp. $a_i \in RL$), we denote f_i by $Reach_{LR}(i, P)$ (resp. $Reach_{RL}(i, P)$). Observe that $Reach_{LR}(i, P)$ is the rightmost point on the line to which the set of i agents at initial positions $Pos[1 : i]$, each having power P, may transport the union of their initial information. Similarly, $Reach_{RL}(i, P)$ is the leftmost such point for agents at positions $Pos[i : n]$.

Lemma 1 permits to construct a linear-time decision procedure verifying if a given amount P of battery power is sufficient to design a convergecast strategy for a given configuration $Pos[1 : n]$ of agents. We first compute two lists $Reach_{LR}(i, P)$, for $1 \leq i \leq n$ and $Reach_{RL}(i, P)$, for $1 \leq i \leq n$. Then we scan them to determine if there exists an index j, such that $Reach_{LR}(j, P) \geq Reach_{RL}(j + 1, P)$. In such a case, we set $LR = \{a_r \mid r \leq j\}$ and $RL = \{a_r \mid r > j\}$ and we apply Lemma 1 to obtain a convergecast strategy where agents a_j and a_{j+1} meet and exchange their information which totals to the entire initial information of the set of agents. If there is no such index j, no convergecast strategy is possible. This implies

Corollary 1. *In $O(n)$ time we can decide if a configuration of n agents on the line, each having a given maximal power P, can perform convergecast.*

The remaining lemmas of this subsection bring up observations needed to construct an $O(n)$ algorithm designing an optimal centralized convergecast strategy.

Note that if the agents are not given enough power, then it can happen that some agent a_p may never learn the information from a_1 (resp. from a_n). In this case, a_p cannot belong to LR (resp. RL). We denote by $Act_{LR}(p)$ the minimum amount of power we have to give the agents to ensure that a_p can learn the information from a_1: if $p > 0$, $Act_{LR}(p) = \min\{P \mid Reach_{LR}(p - 1, P) + P \geq Pos[p]\}$. Similarly, we have $Act_{RL}(p) = \min\{P \mid Reach_{RL}(p + 1, P) - P \leq Pos[p]\}$.

Given a strategy using power P, for each agent $p \in LR$, we have $P \geq Act_{LR}(p)$ and either $Reach_{LR}(p - 1, P) \geq Pos[p]$, or $Reach_{LR}(p - 1, P) \leq Pos[p]$. In the first case, $Reach_{LR}(p, P) = Pos[p] + P$, while in the second case, $Reach_{LR}(p, P) = 2Reach_{LR}(p - 1, P) + P - Pos[p]$.

We define threshold functions $TH_{LR}(p)$ and $TH_{RL}(p)$ that compute for each index p, the minimal amount of agents' power ensuring that agent a_p does not go back when $a_p \in LR$ or $a_p \in RL$ respectively (i.e. such that $b_p = Pos[p]$). For each p, let $TH_{LR}(p) = \min\{P \mid Reach_{LR}(p, P) = Pos[p] + P\}$ and $TH_{RL}(p) = \min\{P \mid Reach_{RL}(p, P) = Pos[p] - P\}$. Clearly, $TH_{LR}(1) = TH_{RL}(n) = 0$.

The next lemma illustrates how to compute $Reach_{LR}(q, P)$ and $Reach_{RL}(q, P)$ if we know $TH_{LR}(p)$ and $TH_{RL}(p)$ for every agent p.

Lemma 2. *Consider an amount of power P and an index q. If $p = \max\{p' \leq q \mid TH_{LR}(p') < P\}$, then $Reach_{LR}(q, P) = 2^{q-p}Pos[p] + (2^{q-p+1} - 1)P - \sum_{i=p+1}^{q} 2^{q-i}Pos[i]$. Similarly, if $p = \min\{p' \geq q \mid TH_{RL}(p') < P\}$, then $Reach_{RL}(q, P) = 2^{p-q}Pos[p] - (2^{p-q+1} - 1)P - \sum_{i=q}^{p-1} 2^{i-q}Pos[i]$.*

Observe that the previous lemma implies that, for each q, the function $Reach_{LR}(q,\cdot)$ is an increasing, continuous, piecewise linear function on $[Act_{LR}(q),+\infty)$ and that $Reach_{RL}(q,\cdot)$ is a decreasing, continuous, piecewise linear function on $[Act_{RL}(q),+\infty)$.

In the following, we denote $S_{LR}(p,q) = \sum_{i=p+1}^{q} 2^{q-i}Pos[i]$ and $S_{RL}(p,q) = \sum_{i=q}^{p-1} 2^{i-q}Pos[i]$.

Remark 1. For every $p \leq q \leq r$, $S_{LR}(p,r) = 2^{r-q}S_{LR}(p,q) + S_{LR}(q,r)$.

We now show that for an optimal convergecast strategy, the last agent of LR and the first agent of RL meet at some point between their initial positions and that they need to use all the available power to meet.

Lemma 3. *Suppose there exists an optimal convergecast strategy for a configuration $Pos[1:n]$, where the maximum power used by an agent is P. Then, there exists an integer $1 \leq p < n$ such that $Pos[p] < Reach_{LR}(p,P) = Reach_{RL}(p+1,P) < Pos[p+1]$.*

Moreover, $\forall q \leq p$, $Act_{LR}(q) < P < TH_{RL}(q)$ and $\forall q > p$, $Act_{RL}(q) < P < TH_{LR}(q)$.

2.2 A Linear Algorithm to Compute the Optimal Power Needed for Convergecast

In this section, we prove the following theorem.

Theorem 1. *An optimal convergecast strategy for the line can be computed in linear time.*

We first explain how to compute a stack of couples $(p, TH_{LR}(p))$that we can subsequently use to compute $Reach_{LR}(p,P)$ for any given P. Then, we present a linear algorithm that computes the value needed to solve convergecast when the last index $r \in LR$ is provided: given an index r, we compute the optimal power needed to solve convergecast assuming that $LR = \{a_q \mid q \leq r\}$ and $RL = \{a_q \mid q > r\}$. Finally, we explain how to use techniques introduced for the two previous algorithms in order to compute the optimal power needed to solve convergecast. These computations directly imply the schedule of the agents' moves of the optimal convergecast strategy.

Computing the Thresholds Values. To describe explicitly the function $Reach_{LR}(q,\cdot)$, we need to identify the indexes p such that for every $r \in [p+1,q]$, we have $TH_{LR}(r) > TH_{LR}(p)$. They correspond to the breakpoints at which the slopes of the piecewise linear function $Reach_{LR}(q,\cdot)$ change. Indeed, if we are given such an index p, then for every P comprised between $TH_{LR}(p)$ and $\min\{TH_{LR}(r) \mid p < r \leq q\}$, we have $Reach_{LR}(q,P) = 2^{q-p}Pos[p] + (2^{q-p+1}-1)P - S_{LR}(p,q)$. We denote by $X_{LR}(q)$ this set of indexes $\{p \leq q \mid \forall r \in [p+1,q], TH_{LR}(r) > TH_{LR}(p)\}$.

In particular, if we want to compute $TH_{LR}(q+1)$, we just need to find $p = \max\{r \leq q \mid Reach_{LR}(q, TH_{LR}(r)) < Pos[q+1]\}$, and then $TH_{LR}(q+1)$ is the value of power P such that $2^{q-p}Pos[p] + (2^{q-p+1}-1)P - S_{LR}(p,q) = Pos[q+1]$. Moreover, by the choice of p, we have $X_{LR}(q+1) = \{r \in X_{LR}(q) \mid r \leq p\} \cup \{q+1\}$.

Using these remarks, the function `ThresholdLR`, having been given an agent index r, returns a stack $\mathtt{TH_{LR}}$ containing couples (p, P) such that $p \in X_{LR}(r)$ and $P = TH_{LR}(p)$. Note that in the stack $\mathtt{TH_{LR}}$, the elements (p, P) are sorted along both components, the largest being on the top of the stack.

The algorithm proceeds as follows. Initially, the stack $\mathtt{TH_{LR}}$ contains only the couple $(1, TH_{LR}(1))$. At each iteration, given the stack corresponding to the index q, in order to compute the stack for the index $q + 1$, we first pop out all elements (p, P) such that $Reach_{LR}(q, P) > Pos[q+1]$. After that, the integer p needed to compute $TH_{LR}(q+1)$ is located on the top of the stack. Finally, the couple $(q+1, TH_{LR}(q+1))$ is pushed on the stack before we proceed with the subsequent index q. At the end of the procedure, we return the stack $\mathtt{TH_{LR}}$ corresponding to the index r.

The number of stack operations performed during the execution of this function is $O(r)$. However, in order to obtain a linear number of arithmetic operations, we need to be able to compute 2^{q-p} and $S_{LR}(p, q)$ in constant time.

In order to compute 2^{q-p} efficiently, we can store the values of 2^i, $i \in [1, n-1]$ in an auxiliary array, that we have precomputed in $O(n)$ time. We cannot precompute all values of $S_{LR}(p, q)$ since this requires calculating $\Theta(n^2)$ values. However, from Remark 1, we know that $S_{LR}(p, q) = S_{LR}(1, q) - 2^{q-p} S_{LR}(1, p)$. Consequently, it is enough to precompute $S_{LR}(1, i)$ for each $i \in [2, n]$. Since $S_{LR}(1, i+1) = 2S_{LR}(1, i) + Pos[i+1]$, this can be done using $O(n)$ arithmetic operations.

```
Function ThresholdLR(array $Pos[1 : n]$ of real; $r$:integer):stack

TH_LR = empty_stack;
push (TH_LR, $(1, 0)$);
for $q = 1$ to $r - 1$ do
    $(p, P)$ = pop(TH_LR) ;                          /* $p = q$ and $P = TH_{LR}(p)$ */
    while $2^{q-p} * Pos[p] + (2^{q-p+1} - 1) * P - S_{LR}(p, q) \geq Pos[q+1]$ do
    $(p, P)$ = pop(TH_LR);
    /* while $Reach_{LR}(q, P) \geq Pos[q+1]$ we consider the next
       element in TH_LR                                                  */
    push (TH_LR, $(p, P)$);
    $Q = (2^{q-p} * Pos[p] - Pos[q+1] - S_{LR}(p, q))/(2^{q-p+1} - 1)$;
    /* $Q$ is the solution of $Reach_{LR}(q, P) = Pos[q+1]$              */
    push (TH_LR, $(q+1, Q)$);
return (TH_LR);
```

Similarly, we can define the function `ThresholdRL (array` $Pos[1 : n]$ `of real,` r`:integer):stack` that returns a stack $\mathtt{TH_{RL}}$ containing all pairs $(q, TH_{RL}(q))$ such that for every $p \in [r, q-1]$, we have $TH_{RL}(p) > TH_{RL}(q)$.

Computing the Optimal Power When LR and RL Are Known. Suppose now that we are given an agent index r and we want to compute the optimal power needed to solve convergecast when $LR = \{a_p \mid p \leq r\}$ and $RL = \{a_q \mid q > r\}$. From Lemma 3, we know that there exists a unique P_{OPT} such that $Reach_{LR}(r, P_{OPT}) = Reach_{RL}(r + 1, P_{OPT})$.

As previously, by Lemma 2, we know that the value of $Reach_{LR}(r, P_{OPT})$ depends on $p = \max\{p' \leq r \mid TH_{LR}(p') < P_{OPT}\}$. Similarly, $Reach_{RL}(r+1, P_{OPT})$ depends on $q = \min\{q' \geq r+1 \mid TH_{RL}(q') < P_{OPT}\}$. If we are given the values of p and q, then P_{OPT} is the value of P such that

$$2^{r-p}Pos[p] - (2^{r-p+1}-1)P - S_{LR}(p,r) = 2^{q-r-1}Pos[q] - (2^{q-r}-1)P - S_{RL}(q,r+1).$$

In Algorithm `OptimalAtIndex`, we first use the previous algorithm to compute the two stacks $\mathtt{TH_{LR}}$ and $\mathtt{TH_{RL}}$ containing respectively $\{(p, TH_{LR}(p)) \mid p \in X_{LR}(r)\}$ and $\{(q, TH_{RL}(q)) \mid q \in X_{RL}(r+1)\}$. Then at each iteration, we consider the two elements (p, P_{LR}) and (q, P_{RL}) that are on top of both stacks. If $P_{LR} \geq P_{RL}$ (the other case is symmetric), we check whether $Reach_{LR}(r, P_{LR}) \geq Reach_{RL}(r+1, P_{LR})$. In this case, we have $P > P_{OPT}$, so we remove (p, P_{LR}) from the stack $\mathtt{TH_{LR}}$ and we proceed to the next iteration. If $Reach_{LR}(r, P_{LR}) < Reach_{RL}(r+1, P_{LR})$, we know that $P_{OPT} \geq P_{LR} \geq P_{RL}$ and we can compute the value of P_{OPT} using Lemma 2.

```
Function OptimalAtIndex(array $Pos[1 : n]$ of real; r:integer):stack
  $\mathtt{TH_{LR}}$ = ThresholdLR($r$);  $\mathtt{TH_{RL}}$ = ThresholdRL($r+1$) ;
  $(p, P_{LR})$ = pop($\mathtt{TH_{LR}}$);  $(q, P_{RL})$ = pop($\mathtt{TH_{RL}}$);  $P = \max\{P_{LR}, P_{RL}\}$;
  /* $p = r$, $P_{LR} = TH_{LR}(r)$, $q = r+1$, $P_{RL} = TH_{RL}(r+1)$.  */
  while
  $2^{r-p}Pos[p]+(2^{r-p+1}-1)P-S_{LR}(p,r) \geq 2^{q-r-1}Pos[q]-(2^{q-r}-1)P-S_{RL}(q,r+1)$
  do                        /* While $Reach_{LR}(r,P) \geq Reach_{RL}(r+1,P)$ do */
      if $P_{LR} \geq P_{RL}$ then  $(p, P_{LR})$ = pop($\mathtt{TH_{LR}}$);
      else  $(q, P_{RL})$ = pop($\mathtt{TH_{RL}}$);
      $P = \max\{P_{LR}, P_{RL}\}$;
  $P_{OPT}$ =
  $(2^{q-r-1}Pos[q] - S_{RL}(q,r+1) - 2^{r-p}Pos[p] + S_{LR}(p,r))/(2^{r-p+1} + 2^{q-r} - 2)$;
  /* $P_{OPT}$ is the solution of
      $Reach_{LR}(r, P_{OPT}) = Reach_{RL}(r+1, P_{OPT})$  */
  return ($P_{OPT}$);
```

Let $Y_{LR}(r, P)$ denote $\{(p, TH_{LR}(p)) \mid p \in X_{LR}(r) \text{ and } TH_{LR}(p) < P\}$ and $Y_{RL}(r+1, P) = \{(q, TH_{RL}(q)) \mid q \in X_{RL}(r+1) \text{ and } TH_{RL}(q) < P\}$.

Remark 2. At the end of the execution of the function `OptimalAtIndex`, $\mathtt{TH_{LR}}$ and $\mathtt{TH_{RL}}$ contain respectively $Y_{LR}(r, P_{OPT})$ and $Y_{RL}(r+1, P_{OPT})$.

Moreover, if initially the two stacks $\mathtt{TH_{LR}}$ and $\mathtt{TH_{RL}}$ contain respectively $Y_{LR}(r, P)$ and $Y_{RL}(r+1, P)$ for some $P \geq P_{OPT}$, then the value computed by the algorithm is also P_{OPT} .

Computing the Optimal Power for convergecast. We now explain how to compute the optimal amount of power needed to achieve convergecast using a linear number of operations.

Let $P_{<r}$ be the optimal value needed to solve convergecast when $\max\{s \mid a_s \in LR\} < r$, i.e., when the two agents whose meeting results in merging the entire information are a_i and a_{i+1} for some $i < r$. If $Reach_{LR}(r, P_{<r}) \leq Reach_{RL}(r+1, P_{<r})$, then $P_{<r+1} = P_{<r}$. However, if $Reach_{LR}(r, P_{<r}) > Reach_{RL}(r+1, P_{<r})$, then $P_{<r+1} < P_{<r}$ and $P_{<r+1}$ is the unique value of P such that $Reach_{LR}(r, P) = Reach_{RL}(r+1, P)$. This corresponds to the value returned by `OptimalAtIndex` (Pos, r).

The general idea of Algorithm `ComputeOptimal` is to iteratively compute the value of $P_{<r}$. If we need a linear time algorithm, we cannot call repeatedly the function `OptimalAtIndex`. However, from Remark 2, in order to compute $P_{<r+1}$ when $P_{<r+1} \leq P_{<r}$, it is enough to know $Y_{LR}(r, P_{<r})$ and $Y_{RL}(r+1, P_{<r})$. If we know $Y_{LR}(r, P_{<r})$ and $Y_{RL}(r+1, P_{<r})$, then we can use the same algorithm as in `OptimalAtIndex` in order to compute $P_{<r+1}$. Moreover, from Remark 2, we also get $Y_{LR}(r, P_{<r+1})$ and $Y_{RL}(r+1, P_{<r+1})$ when we compute $P_{<r+1}$.

Before proceeding to the next iteration, we need to compute $Y_{LR}(r+1, P_{<r+1})$ and $Y_{RL}(r+2, P_{<r+1})$ from $Y_{LR}(r, P_{<r+1})$ and $Y_{RL}(r+1, P_{<r+1})$. Note that if $TH_{LR}(r) > P_{<r+1}$, then $Y_{LR}(r+1, P_{<r+1}) = Y_{LR}(r, P_{<r+1})$. If $TH_{LR}(r) \leq P_{<r+1}$, we can use the same algorithm as in `ThresholdLR` to compute $Y_{LR}(r+1, P_{<r+1}) = \{(p, TH_{LR}(p)) \mid p \in X_{LR}(r)\}$ from $Y_{LR}(r, P_{<r+1})$. Consider now $Y_{RL}(r+2, P_{<r+1})$. If $TH_{RL}(r+1) > P_{<r+1}$, then $(r+1, TH_{RL}(r+1)) \notin Y_{RL}(r+1, P_{<r+1})$, and $Y_{RL}(r+2, P_{<r+1}) = Y_{RL}(r+1, P_{<r+1})$. If $TH_{RL}(r+1) \leq P_{<r+1}$, then either $Pos[r+1] - P_{<r+1} \geq Reach_{RL}(r+1, P_{<r+1})$ if $P_{<r+1} = P_{<r}$, or $Pos[r+1] - P_{<r+1} = Reach_{RL}(r+1, P_{<r+1}) = Reach_{LR}(r, P_{<r+1})$ if $P_{<r+1} < P_{<r}$. In both cases, it implies that $Act_{LR}(r+1) \geq P_{<r+1}$. Therefore, by Lemma 3, $P_{<i} = P_{<r+1}$ for every $i \geq r+1$ and we can return the value of $P_{<r+1}$.

In Algorithm `ComputeOptimal`, at each iteration, the stack $\mathtt{TH_{LR}}$ contains $Y_{LR}(r, P_{<r})$ (except its top element) and the stack $\mathtt{TH_{RL}}$ contains $Y_{RL}(r+1, P_{<r})$ (except its top element). Initially, $\mathtt{TH_{LR}}$ is empty and $\mathtt{TH_{RL}}$ contains $O(n)$ elements. In each iteration, at most one element is pushed into the stack $\mathtt{TH_{LR}}$ and no element is pushed into the stack $\mathtt{TH_{RL}}$. Consequently, the number of stack operations performed by the algorithm is linear.

3 Distributed Convergecast on Trees

A configuration of convergecast on graphs is a couple (G, A) where G is the weighted graph encoding the network and A is the set of the starting nodes of the agents. Let $D(G, A) = \max_{\emptyset \subsetneq X \subsetneq A}\{\min_{x \in X, y \in A \setminus X}\{d_G(x, y)\}\}$ where $d_G(x, y)$ is the distance between x and y in G. Clearly, we have $D(G, A) \leq 2P_{OPT}$.

We consider weighted trees with agents at every leaf. The next theorem states that there exists a 2-competitive distributed algorithm for the convergecast problem on trees.

Theorem 2. *Consider a configuration (T, A) where T is a tree and A contains all the leaves of T. There is a distributed convergecast algorithm using $D(T, A) \leq 2P_{OPT}$ power per agent.*

Sketch of the proof: In order to perform the convergecast, each agent executes Algorithm 1.

Algorithm 1. UnknownTree

```
$collecting$ = true;
while $collecting$ = true do
    Wait until there is at most one port unused by the agent incoming at the current node;
    if all ports of current node were used by incoming agents then
    $collecting$ = false;
    if the agent has used less power than any other agent present at the node and $collecting$ = true then
        Move through the unused incoming port until you meet another agent or reach a node;
    else $collecting$ = false;
    if the agent is inside an edge then $collecting$ = false;
```

The agents traverse the leaf edges then edges between nodes at height one and two and so on. When all the tree is traversed, all the information is collected at the last meeting point. No agent will use more power than $D(T, A) \leq 2P_{OPT}$. □

The following theorem shows that no distributed algorithm may offer a better competitive ratio than 2 even if we only consider line networks.

Theorem 3. *Consider any $\delta > 0$, and any value of power P. There exists an integer n and a configuration $Pos[1 : n]$ of n agents on the line such that there is a convergecast strategy using power P and so that there is no deterministic distributed strategy allowing the agents to solve convergecast when the amount of power given to each agent is $(2 - \delta)P$.*

Sketch of the proof: Let $\varepsilon = \delta P/4$ and $\sigma = \varepsilon/2 = \delta P/8$. Let $l = \lfloor \log(8/\delta) \rfloor$ and $k = l + 2$.

Consider a set of agents positioned on a line as follows (See Figure 1). There is an agent a_0 (resp. a_{2l+1}) at the left (resp. right) end of the line on position $s'_0 = 0$ (resp. $s_{2l+1} = \ell$). For each $1 \leq i \leq 2l$, there is a set A_i of k agents on distinct initial positions within a segment $[s_i, s'_i]$ of length σ such that for each $1 \leq i \leq 2l + 1$, the distance between s_i and s'_{i-1} is $2(P - \varepsilon)$. Using Lemma 2, we can show, that if the amount of power given to each agent is P, then convergecast is achievable.

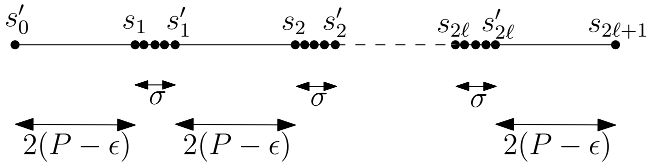


Fig. 1. The configuration in the proof of Theorem 3

Suppose now that there exists a distributed strategy $\mathcal{S}$ that solves convergecast on the configuration when the amount of power given to each agent is $(2 - \delta)P$. We can show that for each $i \in [1, l]$, all agents from A_i perform the same moves as long as the leftmost agent of A_i has not met any agent from A_{i-1}. We show by induction on $i \in [1, l]$ that agents in A_i learn the information from a_0 before the one from a_{2l+1} and that no agent in the set A_i knowing the information from a_0 can reach the point $s_{i+1} - (2^{i+2} - 2)\varepsilon$. If we consider the agents from A_l, we get that no agent from A_{l-1} can reach $s_l - (2^{l+1} - 2)\varepsilon \leq s_l - (8/\delta - 2)\delta P/4 = s_l - 2P + \delta P/2 < s_l - 2P + \delta P$ having the initial information of a_0. Since no agent from the set A_l can reach any point on the left of $s_l - 2P + \delta P$, it implies that no agent from A_l can ever learn the information from a_0 and thus, $\mathcal{S}$ is not a distributed convergecast strategy. □

4 Centralized Convergecast on Trees and Graphs

We show in this section that for trees the centralized convergecast problem is substantially harder than for lines.

Theorem 4. *The centralized convergecast decision problem is strongly NP-complete for trees.*

Sketch of the proof: We construct a polynomial-time many-one reduction from the 3-Partition problem which is strongly NP-Complete [27]. The 3-partition problem answers the following question: can a given multiset S of $3m$ positive integers x_i such that $R/4 < x_i < R/2$ be partitioned into m disjoint sets $S_1, S_2, \ldots, S_m$ of size three such that for $1 \leq j \leq m$, $\sum_{x \in S_j} x = R$? The instance of the centralized convergecast problem constructed from an instance of 3-partition is the star depicted in Figure 2. There is an agent at each leaf and each agent is given power equal to $2R + 1$. We can assume that in any convergecast strategy, agents a_i first move to the center u, each agent b_i moves at distance x_i from u and agent c moves at distance $2R$ from u. Agents a_i, for $1 \leq i \leq m$, must then collect the information from agents b_i and finally agent a_{m+1} must move to c in order to complete the convergecast. Since agents a_i, while reaching u have the remaining power of $2R$ and that collecting information from b_i and return to node u costs $2x_i$ power, the instance of convergecast has a solution if and only if the original instance of 3-partition has a solution.

It remains to show that the problem is in NP. Given a strategy $\mathcal{S}$, the certificate of the instance encodes in chronological order the positions of meetings in which at least one agent learns a new piece of information. There is a polynomial number of such meetings called useful meetings. We can show that the strategy $\mathcal{S}'$, in which agents move via shortest paths to their useful meetings, is a convergecast strategy and uses less power than $\mathcal{S}$. If a useful meeting occurs on an edge, the certificate encodes the name of a variable d_i that represents the exact position inside the edge. Checking that we can assign values to d_i, such that each agent moves a distance less than P in $\mathcal{S}'$, can be done in polynomial time using linear programming. □

Even if the exact centralized optimization problem is NP-complete, we can obtain a 2-approximation of the power needed to achieve centralized convergecast in arbitrary graphs in polynomial time.

Theorem 5. *Consider a configuration (G, A) for an arbitrary graph G. There is a polynomial algorithm computing a centralized convergecast strategy using $D(G, A) \leq 2P_{OPT}$ power per agent.*

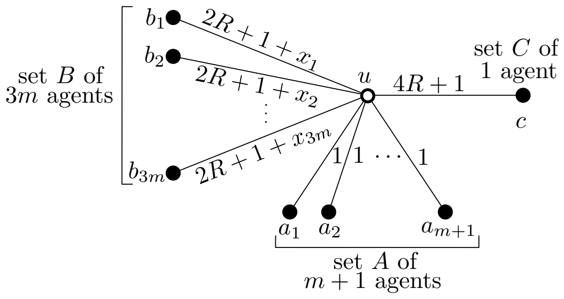


Fig. 2. Instance of centralized convergecast problem from an instance of 3-partition in the proof of Theorem 4

Sketch of the proof: The idea of the algorithm is to construct a total order u_i on the positions of agents in the following way. We put in set V an arbitrary agent's initial position u_1. Iteratively, we add to the set V of already treated agents' positions a new agent's position u_i, which is at the closest distance from V along some path P_i. Then agents move in the reverse order, starting with agent at u_n. Agent at u_i moves following the path P_i. The length of P_i is less than $D(G, A)$. When all agents have moved, the agent at u_1 gets all the information. □

5 Conclusion and Open Problems

It is worth pursuing questions related to information transportation by mobile agents to other communication tasks, such as broadcasting or gossiping. Only some of our techniques on convergecast extend to these settings (e.g. NP-hardness for trees).

The problem of a single *information transfer* by mobile agents between two stationary points of the network is also interesting. In particular, it is an open question whether this problem for tree networks is still NP-hard or if a polynomial-time algorithm is possible, since our reduction to 3-partition is no longer valid.

Other related questions may involve agents with unequal power, agents with non-zero visibility, labeled agents, unreliable agents or networks, etc.

References

1. Albers, S.: Energy-efficient algorithms. Comm. ACM 53(5), 86–96 (2010)
2. Albers, S., Henzinger, M.R.: Exploring unknown environments. SIAM J. on Comput. 29(4),1164–1188
3. Alpern, S., Gal, S.: The theory of search games and rendezvous. Kluwer Academic Publ. (2002)
4. Ambühl, C.: An Optimal Bound for the MST Algorithm to Compute Energy Efficient Broadcast Trees in Wireless Networks. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1139–1150. Springer, Heidelberg (2005)

5. Ando, H., Oasa, Y., Suzuki, I., Yamashita, M.: Distributed memoryless point convergence algorithm for mobile robots with limited visibility. IEEE Trans. on Robotics and Automation 15(5), 818–828 (1999)
6. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. In: Distributed Computing, pp. 235–253 (2006)
7. Annamalai, V., Gupta, S.K.S., Schwiebert, L.: On Tree-Based Convergecasting in Wireless Sensor Networks. IEEE Wireless Communications and Networking 3, 1942–1947 (2003)
8. Augustine, J., Irani, S., Swamy, C.: Optimal powerdown strategies. SIAM J. Comput. 37, 1499–1516 (2008)
9. Averbakh, I., Berman, O.: A heuristic with worst-case analysis for minimax routing of two traveling salesmen on a tree. Discrete Applied Mathematics 68, 17–32 (1996)
10. Azar, Y.: On-line Load Balancing. In: Fiat, A., Woeginger, G. (eds.) Online Algorithms 1996. LNCS, vol. 1442, pp. 178–195. Springer, Heidelberg (1998)
11. Awerbuch, B., Betke, M., Rivest, R., Singh, M.: Piecemeal graph exploration by a mobile robot. Information and Computation 152, 155–172 (1999)
12. Baeza Yates, R.A., Culberson, J.C., Rawlins, G.J.E.: Searching in the Plane. Information and Computation 106(2), 234–252 (1993)
13. Bender, M., Fernandez, A., Ron, D., Sahai, A., Vadhan, S.: The power of a pebble: exploring and mapping directed graphs. In: Proc. 30th STOC, pp. 269–278 (1998)
14. Bender, M., Slonim, D.: The power of team exploration: two robots can learn unlabeled directed graphs. In: Proc. 35th FOCS, pp. 75–85 (1994)
15. Betke, M., Rivest, R.L., Singh, M.: Piecemeal learning of an unknown environment. Machine Learning 18(2/3), 231–254 (1995)
16. Blum, A., Raghavan, P., Schieber, B.: Navigating in unfamiliar geometric terrain. SIAM J. Comput. 26(1), 110–137 (1997)
17. Bunde, D.P.: Power-aware scheduling for makespan and flow. In: SPAA, pp. 190–196 (2006)
18. Chen, F., Johnson, M.P., Alayev, Y., Bar-Noy, A., La Porta, T.F.: Who, When, Where: Timeslot Assignment to Mobile Clients. IEEE Transactions on Mobile Computing 11(1), 73–85 (2012)
19. Cieliebak, M., Flocchini, P., Prencipe, G., Santoro, N.: Solving the Robots Gathering Problem. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 1181–1196. Springer, Heidelberg (2003)
20. Cohen, R., Peleg, D.: Convergence Properties of the Gravitational Algorithm in Asynchronous Robot Systems. SIAM J. on Comput. 34(6), 1516–1528 (2005)
21. Cord-Landwehr, A., Degener, B., Fischer, M., Hüllmann, M., Kempkes, B., Klaas, A., Kling, P., Kurras, S., Märtens, M., Meyer auf der Heide, F., Raupach, C., Swierkot, K., Warner, D., Weddemann, C., Wonisch, D.: A New Approach for Analyzing Convergence Algorithms for Mobile Robots. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 650–661. Springer, Heidelberg (2011)
22. Deng, X., Papadimitriou, C.H.: Exploring an unknown graph. In: Proc. 31st FOCS, vol. I, pp. 355–361 (1990)
23. Das, S., Flocchini, P., Santoro, N., Yamashita, M.: On the Computational Power of Oblivious Robots: Forming a Series of Geometric Patterns. In: Proc. PODC, pp. 267–276 (2010)
24. Dynia, M., Korzeniowski, M., Schindelhauer, C.: Power-Aware Collective Tree Exploration. In: Grass, W., Sick, B., Waldschmidt, K. (eds.) ARCS 2006. LNCS, vol. 3894, pp. 341–351. Springer, Heidelberg (2006)
25. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Gathering of asynchronous robots with limited visibility. Th. Comp. Science 337, 147–168 (2005)
26. Fraigniaud, P., Gąsieniec, L., Kowalski, D.R., Pelc, A.: Collective Tree Exploration. In: Farach-Colton, M. (ed.) LATIN 2004. LNCS, vol. 2976, pp. 141–151. Springer, Heidelberg (2004)

27. Garey, M.R., Johnson, D.S.: Computers and Intractability. A Guide to the Theory of NP-Completeness, 96–105, 224 (1979)
28. Frederickson, G., Hecht, M., Kim, C.: Approximation algorithms for some routing problems. SIAM J. on Comput. 7, 178–193 (1978)
29. Heinzelman, W.B., Chandrakasan, A.P., Balakrishnan, H.: An Application-Specific Protocol Architecture for Wireless Microsensor Networks. Transactions on Wireless Communication 1(4), 660–670 (2002)
30. Irani, S., Shukla, S.K., Gupta, R.: Algorithms for power savings. ACM Trans. on Algorithms 3(4), Article 41 (2007)
31. Kesselman, A., Kowalski, D.R.: Fast distributed algorithm for convergecast in ad hoc geometric radio networks. Journal of Parallel and Distributed Computing 66(4), 578–585 (2006)
32. Krishnamachari, L., Estrin, D., Wicker, S.: The impact of data aggregation in wireless sensor networks. In: ICDCS Workshops, pp. 575–578 (2002)
33. Megow, N., Mehlhorn, K., Schweitzer, P.: Online Graph Exploration: New Results on Old and New Algorithms. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 478–489. Springer, Heidelberg (2011)
34. Nikoletseas, S., Spirakis, P.G.: Distributed Algorithms for Energy Efficient Routing and Tracking in Wireless Sensor Networks. Algorithms 2, 121–157 (2009)
35. Rajagopalan, R., Varshney, P.K.: Data-aggregation techniques in sensor networks: a survey. IEEE Communications Surveys and Tutorials 8(4), 48–63 (2006)
36. Stojmenovic, I., Lin, X.: Power-Aware Localized Routing in Wireless Networks. IEEE Trans. Parallel Distrib. Syst. 12(11), 1122–1133 (2001)
37. Suzuki, I., Yamashita, M.: Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. SIAM J. Comput. 28(4), 1347–1363 (1999)
38. Yamashita, M., Suzuki, I.: Characterizing geometric patterns formable by oblivious anonymous mobile robots. Th. Comp. Science 411(26-28), 2433–2453 (2010)
39. Yao, F.F., Demers, A.J., Shenker, S.: A scheduling model for reduced CPU energy. In: Proc. of 36th FOCS, pp. 374–382 (1995)

Memory Lower Bounds for Randomized Collaborative Search and Implications for Biology

Ofer Feinerman[1,⋆] and Amos Korman[2,⋆⋆]

[1] Incumbent of the The Louis and Ida Rich Career Development Chair, The Weizmann Institute of Science, Rehovot, Israel
feiner@weizmann.ac.il

[2] CNRS and University Paris Diderot, France
amos.korman@liafa.jussieu.fr

Abstract. Initial knowledge regarding group size can be crucial for collective performance. We study this relation in the context of the *Ants Nearby Treasure Search (ANTS)* problem [18], which models natural cooperative foraging behavior such as that performed by ants around their nest. In this problem, k (probabilistic) agents, initially placed at some central location, collectively search for a treasure on the two-dimensional grid. The treasure is placed at a target location by an adversary and the goal is to find it as fast as possible as a function of both k and D, where D is the (unknown) distance between the central location and the target. It is easy to see that $T = \Omega(D + D^2/k)$ time units are necessary for finding the treasure. Recently, it has been established that $O(T)$ time is sufficient if the agents know their total number k (or a constant approximation of it), and enough memory bits are available at their disposal [18]. In this paper, we establish lower bounds on the agent memory size required for achieving certain running time performances. To the best our knowledge, these bounds are the first non-trivial lower bounds for the memory size of probabilistic searchers. For example, for every given positive constant ϵ, terminating the search by time $O(\log^{1-\epsilon} k \cdot T)$ requires agents to use $\Omega(\log\log k)$ memory bits.

From a high level perspective, we illustrate how methods from distributed computing can be useful in generating lower bounds for cooperative biological ensembles. Indeed, if experiments that comply with our setting reveal that the ants' search is time efficient, then our theoretical lower bounds can provide some insight on the memory they use for this task.

1 Introduction

Background and Motivation: In biology, individuals assemble into groups that allow them, among other things, to monitor and react to relatively large

⋆ Supported by the THE ISRAEL SCIENCE FOUNDATION FIRST grant (No. 1694/10) and by the Clore Foundation.

⋆⋆ Supported in part by the ANR projects DISPLEXITY and PROSE, and by the INRIA project GANG.

environments. For this, the individuals typically spread over length scales that are much larger than those required for communication. Thus, collecting knowledge regarding large areas dictates a dispersion that may come at the price of group coordination. A possible solution involves the use of designated, localized areas where individuals convene to share information and from which they then disperse to interact with the environment. Indeed, such convention areas have been identified in a wide spectrum of biological systems ranging from groups of immuno-cells [33] to flocking birds [16,54].

Here we focus, on a third example, that of collective *central place foraging* [32,40] where a group of animals leave a central location (*e.g.,* a nest) to which they then retrieve collected food items. Ants, for example, were shown to communicate within their nest regarding food availability and quality outside it [9,29]. This information is then used as a means of regulating foraging efforts. It may be the case that such communication is constrained to the central place alone. For example, once they leave their nest, desert ants (*e.g.,* of the genus *Cataglyphys*) rarely interact [32]. This is due to their dispersedness and lack of chemical trail markings. Similar communication constraints are also experienced by the honeybee *Apis mellifera*. Other than being central-place, the search we discuss is cooperative: although it is conducted by individuals any findings are shared.

One piece of information that may be available to a localized group is its *size*. In a process known as *quorum sensing*, a threshold estimate of group size is used to reach collective decisions and choose between divergent courses of action [10,17,43,50,51]. Here, we study the potential benefits of estimating group size in the context of collective central place foraging without mid-search interactions. Clearly, due to competition and other time constrains, food items must be found relatively fast. Furthermore, finding food not only fast but also in proximity to the central location holds numerous advantages at both the search and the retrieval stages [32,38,40]. Intuitively, the problem at hand is efficiently distributing searchers within bounded areas around the nest while minimizing overlaps.

It was previously shown that the efficiency of collective central place foraging may be enhanced by initial knowledge regarding group size [18]. More specifically, that paper introduces the *Ants Nearby Treasure Search (ANTS)* problem, which models the aforementioned central place foraging setting. In this problem, k (probabilistic) agents, initially placed at some central location, collectively search for a treasure in the two-dimensional grid. The treasure is placed at a target location by an adversary and the goal is to find it as fast as possible as a function of both k and D, where D is the (unknown) distance between the central location and the target. Once the agents initiate the search they cannot communicate between themselves. Based on volume considerations, it is an easy observation that the expected running time of any algorithm is $\Omega(D + D^2/k)$. It was established in [18] that the knowledge of a constant approximation of k allows the agents to find the treasure in asymptotically optimal expected time, namely, $O(D + D^2/k)$. On the other hand, the lack of any information of k prevents them from reaching expected time that is higher than optimal by a factor slightly larger than $O(\log k)$. That work also establishes lower bounds on the competitiveness of the algorithm in the particular case where some given approximation to k is available to all nodes.

In this work, we simulate the initial step of information sharing (*e.g.,* regarding group size) within the nest by using the abstract framework of *advice* (see, *e.g.,* [12,23,25]). That is, we model the preliminary process for gaining knowledge about k (*e.g.,* at the central location) by means of an *oracle* that assigns advice to agents. To measure the amount of information accessible to agents, we analyze the *advice size*, that is, the maximum number of bits used in an advice. Since we are mainly interested in lower bounds on the advice size required to achieve a given competitive ratio, we apply a liberal approach and assume a highly powerful oracle. More specifically, even though it is supposed to model a distributed (probabilistic) process, we assume that the oracle is a centralized probabilistic algorithm (almost unlimited in its computational power) that can assign different agents different advices. Note that, in particular, by considering identifiers as part of the advice, our model allows to relax the assumption that all agents are identical and to allow agents to be of several types. Indeed, in the context of ants, it has been established that ants on their first foraging bouts execute different protocols than those that are more experienced [52].

The main technical results of this paper deal with lower bounds on the advice size. For example, with the terminology of advice, [18] showed that advice of size $O(\log\log k)$ bits is sufficient to obtain an $O(1)$-competitive algorithm. We prove that this bound is tight. In fact, we show a much stronger result, that is, that advice of size $\Omega(\log\log k)$ is necessary even for achieving competitiveness which is as large as $O(\log^{1-\epsilon} k)$, for every given positive constant ϵ. On the other extremity, we show that $\Omega(\log\log\log k)$ bits of advice are necessary for being $O(\log k)$-competitive, and that this bound is tight. In addition, we exhibit lower bounds on the corresponding advice size for a range of intermediate competitivenesses.

Observe that the advice size bounds from below the number of memory bits used by an agent, as this amount of bits in required merely for storing some initial information.

In general, from a purely theoretical point of view, analyzing the memory required for efficient search is a central theme in computer science [45], and is typically considered to be difficult. To the best of our knowledge, the current paper is the first paper establishing non-trivial lower bounds for the memory of randomized searching agents with respect to given time constrains.

From a high level perspective, we illustrate that distributed computing can potentially provide a novel and efficient methodology for the study of highly complex, cooperative biological ensembles. Indeed, if experiments that comply with our setting reveal that the ants' search is time efficient, in the sense detailed above, then our theoretical results can provide some insight on the memory ants use for this task. A detailed discussion of this novel approach is given in Section 5.

Our Results: The main technical results deal with lower bounds on the advice size. Our first result is perhaps the most surprising one. It says not only that

$\Omega(\log\log k)$ bits of advice are required to obtain an $O(1)$-competitive algorithm, but that roughly this amount is necessary even for achieving competitiveness which is as large as $O(\log^{1-\epsilon} k)$, for every given positive constant ϵ. This result should be put in contrast to the fact that with no advice at all, one can obtain a search algorithm whose competitiveness is slightly higher than logarithmic [18].

Theorem 1. *There is no search algorithm that is $O(\log^{1-\epsilon} k)$-competitive for some fixed positive ϵ, using advice of size $o(\log\log k)$.*

On the other extremity, we show that $\Omega(\log\log\log k)$ bits of advice are necessary for constructing an $O(\log k)$-competitive algorithm, and we prove that this bound on the advice is in fact tight.

Theorem 2. *There is no $O(\log k)$-competitive search algorithm, using advice of size $\log\log\log k - \omega(1)$. On the other hand, there exists an $O(\log k)$-competitive search algorithm using advice of size $\log\log\log k + O(1)$.*

Finally, we also exhibit lower bounds for the corresponding advice size for a range of intermediate competitivenesses.

Theorem 3. *Consider a $\Phi(k)$-competitive search algorithm using advice of size $\Psi(k)$. Then, $\Phi(k) = \Omega(\log k/2^{\Psi(k)})$, or in other words, $\Psi(k) = \log\log k - \log\Phi(k) - O(1)$. In particular, if $\Phi(k) = \frac{\log k}{2^{\log^\epsilon \log k}}$, then $\Psi(k) = \log^\epsilon \log k - O(1)$.*

Our results on the advice complexity are summarized in Table 1. As mentioned, our lower bounds on the advice size are also lower bounds on the memory size of agents.

Table 1. Bounds on the advice for given competitiveness

	Competitiveness		Advice size
Tight bound	$O(1)$		$\Theta(\log\log k)$
Tight bound	$O(\log^{1-\epsilon} k)$	$0 < \epsilon < 1$	$\Theta(\log\log k)$
Lower bound	$\log k/2^{\log^\epsilon \log k}$	$0 < \epsilon < 1$	$\log^\epsilon \log k - O(1)$
Tight bound	$O(\log k)$		$\log\log\log k + \Theta(1)$
Upper bound [18]	$O(\log^{1+\epsilon} k)$		zero

Related Work: Our current work falls within the framework of natural algorithms, a recent attempt to study biological phenomena from an algorithmic perspective [1,8,11,18].

The notion of advice is central in computer science. In particular, the concept of advice and its impact on various computations has recently found various applications in distributed computing. In this context, the main measure used is the advice size. It is for instance analyzed in frameworks such as proof labeling [36,37], broadcast [23], local computation of MST [25], graph coloring [24] and graph searching by a single robot [12,31]. Very recently, it has also been investigated in the context of online algorithms [15].

Collective search is a classical problem that has been extensively studied in different contexts (for a more comprehensive summary refer to [18]). Social foraging theory [28] and central place foraging typically deal with optimal resource exploitation strategies between competing or cooperating individuals. Actual collective search trajectories of non-communicating agents have been studied in the physics literature (*e.g.,* [6,46]). Reynolds [46] achieves optimal speed up through overlap reduction which is obtained by sending searchers on near -straight disjoint lines to infinity. This must come at the expense of finding proximal treasures. Harkness and Maroudas [32] combined field experiments with computer simulations of a semi-random collective search and suggest substantial speed ups as group size increases. The collective search problem has further been studied from an engineering perspective (*e.g.,* [42]). In this case, the communication between agents (robots) or their computational abilities are typically unrestricted. These works put no emphasis on finding nearby treasures fast. Further, there is typically (with the exception of [32]) no reference to group size.

In the theory of computer science, the exploration of graphs using mobile agents (or robots) is a central question. (For a more detailed survey refer to *e.g.,* [2,18,20].) Most graph exploration research in concerned with the case of a single deterministic agent exploring a finite graph, see for example [3,13,14,27,41,45]. The more complex situation of multiple identical deterministic agents was studied in [4,20,21,22]. In general, one of the main challenges in search problems is the establishment of memory bounds. For example, the question of whether a single agent can explore all finite undirected graphs using logarithmic memory was open for a long time; answering it to the affirmative [45] established an equality between the classes of languages SL and L.

Evaluating the running time as a function of D, the distance to the treasure, was studied in the context of the cow-path problem [5,34]. In [39], the cow-path problem was extended by considering k agents. However, in contrast to our setting, the agents they consider have unique identities, and the goal is achieved by (centrally) specifying a different path for each of the k agents.

The question of how important it is for individual processors to know their total number has recently been addressed in the context of locality. Generally speaking, it has been observed that for several classical local computation tasks, knowing the number of processors is not essential [35]. On the other hand, in the context of local distributed decision, some evidence exist that such knowledge is crucial for non-deterministic verification [26].

2 Preliminaries

General Setting: We consider the *Ants Nearby Treasure Search (ANTS)* problem initially introduced in [18]. In this *central place* searching problem, k mobile *agents* are searching for a *treasure* on the two-dimensional plane. The agents are probabilistic mobile machines (robots). They are identical, that is, all agents execute the same protocol $\mathcal{P}$. Each agent has some limited field of view, i.e., each agent can see its surrounding up to a distance of some $\varepsilon > 0$. Hence, for simplicity, instead of considering the two-dimensional plane, we assume that the agents are actually walking on the integer two-dimensional infinite grid $G = \mathbb{Z}^2$ (they can traverse an edge of the grid in both directions). The search is central place, that is, all k agents initiate the search from some central node $s \in G$, called the *source*. Before the search is initiated, an adversary locates the treasure at some node $t \in G$, referred to as the *target* node. Once the search is initiated, the agents cannot communicate among themselves[1]. We denote by D the (Manhattan) distance between the source node and the target, i.e., $D = d_G(s,t)$. It is important to note that the agents have no a priori information about the location of t or about D. We say that the agents *find* the treasure when one of the agents visits the target node t. The goal of the agents it to find the treasure as fast as possible as a function of both D and k.

Since we are mainly interested in lower bounds, we assume a very liberal setting. In particular, we do not restrict neither the computational power nor the navigation capabilities of agents. Moreover, we put no restrictions on the internal storage used for navigation. (On the other hand, we note that for constructing upper bounds, the algorithms we consider use simple procedures that can be implemented using relatively little resources.) In addition, even though it may seem natural to require that agents return to the source occasionally (*e.g.,* to know whether other agents have already found the treasure), our lower bounds do not rely on such an assumption.

Oracles and Advice: We would like to model the situation in which before the search actually starts, some initial communication may be made between the agents at the source node. In reality, this preliminary communication may be quite limited. This may be because of difficulties in the communication that are inherent to the agents or the environment, *e.g.,* due to faults or limited memory, or because of asynchrony issues regarding the different starting times of the search, or simply because agents are identical and it may be difficult for agents to distinguish one agent from the other. Nevertheless, we consider a very liberal setting in which this preliminary communication is almost unrestricted.

More specifically, we consider a centralized algorithm called *oracle* that assigns advices to agents in a preliminary stage. The oracle, denoted by $\mathcal{O}$, is a probabilistic centralized algorithm that receives as input a set of k agents and assigns an *advice* to each of the k agents. We assume that the oracle may use a different protocol for each k; given k, the randomized algorithm used for assigning the advices to the k agents is denoted by $\mathcal{O}_k$. An example for such an oracle is the case where each agent is given the same (deterministic) advice which encodes some approximation to the number of agents[2]. However, in principle, the oracle may assign a different advice to each agent, and these advices may not

[1] As mentioned earlier, the particular kinds of animals that we are interested in (*e.g.,* the desert ants *Cataglyphys* and the honeybees *Apis mellifera*) rarely interact outside their source while searching [32]. Nevertheless, the theoretical question of how and to what extent mid-search interactions may aid the search is interesting and a subject of future work.

[2] These types of simple oracles are actually used by our upper bound constructions.

necessarily correspond to an approximation of k. Observe, this definition of an oracle allows it to simulate almost any reasonable preliminary communication between the agents[3].

It is important to stress that even though all agents execute the same searching protocol, they may start the search with different advices. Hence, since their searching protocol may rely on the content of this initial advice, agents with different advices may behave differently. Another important remark concerns the fact that some part of the advices may be used for encoding (not necessarily disjoint) identifiers. That is, assumptions regarding the settings in which not all agents are identical and there are several types of agents can be captured by our setting of advice.

Finally, note that, in contrast to previous works regarding oracles, here, the knowledge of the oracle is limited, as it does not know D. This means, in particular, that a probabilistic oracle may potentially be strictly more powerful than a deterministic one. Indeed, the oracle assigning the advice may use the randomization to reduce the size of the advices by balancing between the efficiency of the search for small values of D and larger values.

To summarize, a *search algorithm* is a pair $\langle \mathcal{P}, \mathcal{O} \rangle$ consisting of a randomized searching protocol $\mathcal{P}$ and randomized oracle $\mathcal{O} = \{\mathcal{O}_k\}_{k \in \mathbf{N}}$. Given k agents, the randomized oracle $\mathcal{O}_k$ assigns a separate advice to each of the given agents. Subsequently, all agents initiate the actual search by letting each of the agents execute protocol $\mathcal{P}$ and using the corresponding advice as input to $\mathcal{P}$. Once the search is initiated, the agents cannot communicate among themselves.

Consider an oracle $\mathcal{O}$. Given k, let $\Psi_{\mathcal{O}}(k)$ denote the maximum number of bits devoted for encoding the advice of an agent, taken over all coin tosses of $\mathcal{O}_k$, and over the k agents. In other words, $\Psi_{\mathcal{O}}(k)$ is the minimum number of bits necessary for encoding the advice, assuming the number of agents is k. Note that $\Psi_{\mathcal{O}}(k)$ also bounds from below the number of memory bits of an agent required by the search algorithm $\langle \mathcal{P}, \mathcal{O} \rangle$, assuming that the number of agents is k. The function $\Psi_{\mathcal{O}}(\cdot)$ is called the *advice size* function of oracle $\mathcal{O}$. (When the context is clear, we may omit the subscript $\mathcal{O}$ from $\Psi_{\mathcal{O}}(\cdot)$ and simply use $\Psi(\cdot)$ instead.)

Time Complexity: When measuring the time to find the treasure, we assume that all internal computations are performed in zero time. For the simplicity of presentation, we assume that the movements of agents are synchronized, that is, each edge traversal is performed in precisely one unit of time. Indeed, this assumption can easily be removed if we measure the time according to the slowest

[3] For example, it can simulate to following very liberal setting. Assume that in the preprocessing stage, the k agents are organized in a clique topology, and that each agent can send a separate message to each other agent. Furthermore, even though the agents are identical, in this preprocessing stage, let us assume that agents can distinguish the messages received from different agents, and that each of the k agents may use a different probabilistic protocol for this preliminary communication. In addition, no restriction is made neither on the memory and computation capabilities of agents nor on the preprocessing time, that is, the preprocessing stage takes finite, yet unlimited, time.

edge-traversal. We also assume that all agents start the search simultaneously at the same time. This assumption can also be easily removed by starting to count the time when the last agent initiates the search.

The *expected running time* of a search algorithm $\mathcal{A} := \langle \mathcal{P}, \mathcal{O} \rangle$ is the expected time until at least one of the agents finds the treasure. The expectation is defined with respect to the coin tosses made by the (probabilistic) oracle $\mathcal{O}$ assigning the advices to the agents, as well as the subsequent coin tosses made by the agents executing $\mathcal{P}$. We denote the expected running time of an algorithm $\mathcal{A}$ by $\tau = \tau_{\mathcal{A}}(D, k)$. In fact, for our lower bound to hold, it is sufficient to assume that the probability that the treasure is found by time 2τ is at least $1/2$. By Markov inequality, this assumption is indeed weaker than the assumption that the expected running time is τ.

Note that if an agent knows D, then it can potentially find the treasure in time $O(D)$, by walking to a distance D in some direction, and then performing a circle around the source of radius D (assuming, of course, that its navigation abilities enable it to perform such a circle). On the other hand, with the absence of knowledge about D, an agent can find the treasure in time $O(D^2)$ by performing a spiral search around the source (see, *e.g.,* [5]). The following observation imply that $\Omega(D + D^2/k)$ is a lower bound on the expected running time of any search algorithm. The proof is straightforward and can be found in [18].

Observation 4. *The expected running time of any algorithm is $\Omega(D + D^2/k)$, even if the number of agents k is known to all agents.*

We evaluate the time performance of an algorithm with respect to the lower bound given by Observation 4. Formally, let $\Phi(k)$ be a function of k. A search algorithm $\mathcal{A} := \langle \mathcal{P}, \mathcal{O} \rangle$ is called $\Phi(k)$-*competitive* if $\tau_{\mathcal{A}}(D, k) \leq \Phi(k) \cdot (D + D^2/k)$, for every integers k and D. Our goal is establish connections between the *size* of the advice, namely $\Psi(k)$, and the competitiveness $\Phi(k)$ of the search algorithm.

More Definitions: The *distance* between two nodes $u, v \in G$, denoted $d(u, v)$, is simply the Manhattan distance between them, i.e., the number of edges on the shortest path connecting u and v in the grid G. For a node u, let $d(u) := d(u, s)$ denote the distance between u and the source node. Hence, $D = d(t)$.

3 Lower Bounds on the Advice

The theorem below generalizes Theorem 4.1 in [18], taking into account the notion of advice. All our lower bound results follow as corollaries of this theorem. Note that for the theorem to be meaningful we are interested in advice size whose order of magnitude is less than $\log \log k$. Indeed, if $\Psi(k) = \log \log k$, then one can encode a 2-approximation of k in each advice, and obtain an optimal result, that is, an $O(1)$-competitive algorithm (see [18]).

Before stating the theorem, we need the following definition. A non-decreasing function $\Phi(x)$ is called *relatively-slow* if $\Phi(x)$ is sublinear (i.e., $\Phi(x) = o(x)$) and if there exist two positive constants c_1 and $c_2 < 2$ such that when restricted to $x > c_1$, we have $\Phi(2x) < c_2 \cdot \Phi(x)$. Note that this definition captures many natural sublinear functions[4].

Theorem 5. *Consider a $\Phi(k)$-competitive search algorithm using advice of size $\Psi(k)$. Assume that $\Phi(\cdot)$ is relatively-slow and that $\Psi(\cdot)$ is non-decreasing. Then there exists some constant x', such that for every $k > 2^{x'}$, the sum $\sum_{i=x'}^{\log k} \frac{1}{\Phi(2^i)\cdot 2^{\Psi(k)}}$ is at most some fixed constant.*

Proof. Consider a search algorithm with advice size $\Psi(k)$ and competitiveness $\Phi'(k)$, where $\Phi'(\cdot)$ is relatively-slow. By definition, the expected running time is less than $\tau(D,k) = (D + D^2/k)\cdot\Phi'(k)$. Note, for $k \leq D$, we have $\tau(D,k) \leq \frac{D^2\Phi(k)}{k}$, where $\Phi(k) = 2\Phi'(k)$, and $\Phi(\cdot)$ is relatively-slow. Let c_1 be the constant promised by the fact that Φ is relatively-slow. Let $x_0 > c_1$ be sufficiently large so that x_0 is a power of 2, and for every $x > x_0$, we have $\Phi(x) < x$ (recall, Φ is sublinear).

Fix an integer $T > x_0^2$. In the remaining of the proof, we assume that the treasure is placed somewhere at distance $D := 2T + 1$. Note, this means, in particular, that by time $2T$ the treasure has not been found yet.

Fix an integer i in $[\log x_0, \frac{1}{2}\log T]$, set $d_i = \sqrt{\frac{T\cdot k_i}{\Phi(k_i)}}$, and let $B(d_i) := \{v \in G : d(v) \leq d_i\}$ denote the ball of radius d_i around the source node. We consider now the case where the algorithm is executed with $k_i := 2^i$ agents (using the corresponding advices given by the oracle $\mathcal{O}_{k_i}$). For every set of nodes $S \subseteq B(d_i)$, let $\chi_i(S)$ denote the random variable indicating the number of nodes in S that were visited by at least one of the k_i agents by time $2T$. (For short, for a singleton node u, we write $\chi_i(u)$ instead of $\chi_i(\{u\})$.) Note, the value of $\chi_i(S)$ depends on the values of the coins tosses made by the oracle for assigning the advices as well as on the values of the coins tossed by the k_i agents. Now, define the ring

$$R_i := B(d_i) \setminus B(d_{i-1}).$$

Claim. For each integer $i \in [\log x_0, \frac{1}{2}\log T]$, we have $\mathbf{E}(\chi_i(R_i)) = \Omega(d_i^2)$.

To see why the claim holds, note that by the properties of Φ, and from the fact that $2^i \leq \sqrt{T}$, we get that $k_i \leq d_i$, and therefore, $\tau(d_i, k_i) \leq \frac{d_i^2\Phi(k_i)}{k_i} = T$. It follows that for each node $u \in B(d_i)$, we have $\tau(d(u), k_i) \leq T$, and hence, the probability that u is visited by time $2T$ is at least $1/2$, that is, $\mathbf{Pr}(\chi_i(u) = 1) \geq 1/2$. Hence, $\mathbf{E}(\chi_i(u)) \geq 1/2$. Now, by linearity of expectation, $\mathbf{E}(\chi_i(R_i)) = \sum_{u\in R_i}\mathbf{E}(\chi_i(u)) \geq |R_i|/2$. Consequently, by time $2T$, the expected number of nodes in R_i that are visited by the k_i agents is $\Omega(|R_i|) = \Omega\left(d_{i-1}(d_i - d_{i-1})\right) = \Omega\left(\frac{T\cdot k_i}{\Phi(k_{i-1})}\cdot\left(\sqrt{\frac{2\Phi(k_{i-1})}{\Phi(k_i)}} - 1\right)\right) = \Omega\left(\frac{T\cdot k_i}{\Phi(k_i)}\right) = \Omega(d_i^2)$, where the second equality follows from the fact that $d_i = d_{i-1}\cdot\sqrt{\frac{2\Phi(k_{i-1})}{\Phi(k_i)}}$, and the third equality follows from the fact that $\Phi(\cdot)$ is relatively-slow. This establishes the claim.

[4] For example, note that the functions of the form $\alpha_0 + \alpha_1\log^{\beta_1} x + \alpha_2\log^{\beta_2}\log x + \alpha_3 2^{\log^{\beta_3}\log x}\log x + \alpha_4\log^{\beta_4} x\log^{\beta_5}\log x$, (for non-negative constants α_i and β_i, $i = 1,2,3,4,5$ such that $\sum_{i=1}^{4}\alpha_i > 0$) are all relatively-slow.

Note that for each $i \in [\log x_0 + 1, \frac{1}{2}\log T]$, the advice given by the oracle to any of the k_i agents must use at most $\Psi(k_i) \leq \Psi(\sqrt{T})$ bits. In other words, for each of these k_i agents, each advice is some integer whose value is at most $2^{\Psi(\sqrt{T})}$.

Let $W(j,i)$ denote the random variable indicating the number of nodes in R_i visited by the j'th agent by time $2T$, assuming that the total number of agents is k_i. By Claim 3, for every integer $i \in [\log x_0 + 1, \frac{1}{2}\log T]$, we have: $\mathbf{E}(\sum_{j=1}^{k_i} W(j,i)) \geq \mathbf{E}(\chi_i(R_i)) = \Omega(d_i^2)$. By linearity of expectation, it follows that for every integer $i \in [\log x_0 + 1, \frac{1}{2}\log T]$, there exists an integer $j \in \{1, 2, \cdots, k_i\}$ for which $\mathbf{E}(W(j,i)) = \Omega(d_i^2/k_i) = \Omega(T/\Phi(k_i))$.

Now, for each advice in the relevant range, i.e., for each $a \in \{1, \cdots, 2^{\Psi(\sqrt{T})}\}$, let $M(a,i)$ denote the random variable indicating the number of nodes in R_i that an agent with advice a visits by time $2T$. Note, the value of $M(a,i)$ depends only on the values of the coin tosses made by the agent. On the other hand, note that the value of $W(j,i)$ depends on the results of the coin tosses made by the oracle assigning the advice, and the results of the coin tosses made by the agent that uses the assigned advice. Recall, the oracle may assign an advice to agent j according to a distribution that is different than the distributions used for other agents. However, regardless of the distribution used by the oracle for agent j, it must be the case that there exists an advice $a_i \in \{1, \cdots, 2^{\Psi(\sqrt{T})}\}$, for which $\mathbf{E}(M(a_i,i)) \geq \mathbf{E}(W(j,i))$. Hence, we obtain:

$$\mathbf{E}(M(a_i,i)) = \Omega(T/\Phi(k_i)).$$

Let $A = \{a_i \mid i \in [\log x_0 + 1, \frac{1}{2}\log T]\}$. Consider now an "imaginary" scenario[5] in which we execute the search algorithm with $|A|$ agents, each having a different advice in A. That is, for each advice $a \in A$, we have a different agent executing the algorithm using advice a. For every set S of nodes, let $\hat{\chi}(S)$ denote the random variable indicating the number of nodes in S that were visited by at least one of these $|A|$ agents (in the "imaginary" scenario) by time $2T$. Let $\hat{\chi} := \hat{\chi}(G)$ denote the random variable indicating the total number of nodes that were visited by at least one of these agents by time $2T$.

By definition, for each $i \in [\log x_0 + 1, \frac{1}{2}\log T]$, the expected number of nodes in R_i visited by at least one of these $|A|$ agents is $\mathbf{E}(\hat{\chi}(R_i)) \geq \mathbf{E}(M(a_i,i)) = \Omega(T/\Phi(k_i))$. Since the sets R_i are pairwise disjoint, the linearity of expectation implies that the expected number of nodes covered by these agents by time $2T$ is $\mathbf{E}(\hat{\chi}) \geq \sum_{i=x_0+1}^{\frac{1}{2}\log T} \mathbf{E}(\hat{\chi}(R_i)) = \Omega\left(\sum_{i=x_0+1}^{\frac{1}{2}\log T} \frac{T}{\Phi(k_i)}\right) = T \cdot \Omega\left(\sum_{i=x_0+1}^{\frac{1}{2}\log T} \frac{1}{\Phi(2^i)}\right)$. Recall that A is included in $\{1, \cdots, 2^{\Psi(\sqrt{T})}\}$. Hence, once more by linearity of expectation, there must exist an advice $\hat{a} \in A$, such that the expected number of nodes

[5] The scenario is called imaginary, because, instead of letting the oracle assign the advice for the agents, we impose a particular advice to each agent, and let the agents perform the search with our advices. Note, even though such a scenario cannot occur by the definition of the model, each individual agent with advice a cannot distinguish this case from the case that the number of agents was some k' and the oracle assigned it the advice a.

that an agent with advice $\hat{a}$ visits by time $2T$ is $T \cdot \Omega\left(\sum_{i=x_0+1}^{\frac{1}{2}\log T} \frac{1}{\Phi(2^i)\cdot 2^{\Psi(\sqrt{T})}}\right)$. Since each agent may visit at most one node in one unit of time, it follows that, for every T large enough, the sum $\sum_{i=x_0+1}^{\frac{1}{2}\log T} 1/\Phi(2^i)\cdot 2^{\Psi(\sqrt{T})}$ is at most some fixed constant. The proof of the theorem now follows by replacing the variable T with T^2. □

Corollary 1. *Consider a $\Phi(k)$-competitive search algorithm using advice of size $\Psi(k)$. Assume that $\Phi(\cdot)$ is relatively-slow. Then, $\Phi(k) = \Omega(\log k/2^{\Psi(k)})$, or in other words, $\Psi(k) = \log\log k - \log \Phi(k) - O(1)$.*

Proof. Theorem 5 says that for every k, we have $\frac{1}{2^{\Psi(k)}}\sum_{i=1}^{\log k}\frac{1}{\Phi(2^i)} = O(1)$. On the other hand, since Φ is non-decreasing, we have $\sum_{i=1}^{\log k}\frac{1}{\Phi(2^i)} \geq \frac{\log k}{\Phi(k)}$. Hence, $\frac{\log k}{2^{\Psi(k)}\cdot\Phi(k)} = O(1)$. The corollary follows. □

The following corollary follows directly from the previous one.

Corollary 2. *Let $\epsilon < 1$ be a positive constant. Consider a $\frac{\log k}{2^{\log^{\epsilon}\log k}}$-competitive search algorithm using advice of size $\Psi(k)$. Then $\Psi(k) = \log^{\epsilon}\log k - O(1)$.*

Our next corollary implies that even though $O(\log\log k)$ bits of advice are sufficient for obtaining $O(1)$-competitiveness, roughly this amount of advice is necessary even for achieving relatively large competitiveness.

Corollary 3. *There is no $O(\log^{1-\epsilon} k)$-competitive search algorithm for some positive constant ϵ, using advice of size $\Psi(k) = \epsilon\log\log k - \omega(1)$.*

Proof. Assume that the competitiveness is $\Phi(k) = O(\log^{1-\epsilon} k)$. Then, we have $\sum_{i=1}^{\log k}\frac{1}{\Phi(2^i)\cdot 2^{\Psi(k)}} = \Omega\left(\frac{\log^{\epsilon} k}{2^{\Psi(k)}}\right)$. According to Theorem 5, this sum is constantly bounded, and hence, we cannot have $\Psi(k) = \epsilon\log\log k - \omega(1)$. □

Corollary 4. *There is no $O(\log k)$-competitive search algorithm, using advice of size $\log\log\log k - \omega(1)$.*

Proof. Assume that the competitiveness is $\Phi(k) = O(\log k)$. Since $\Phi(2^i) = O(i)$, we have $\sum_{i=1}^{\log k} 1/\Phi(2^i) = \sum_{i=1}^{\log k} 1/i = \Omega(\log\log k)$. According to Theorem 5, $\frac{1}{2^{\Psi(k)}}\sum_{i=1}^{\log k} 1/\Phi(2^i) = \Omega(\log\log k/2^{\Psi(k)})$ must converge as k goes to infinity. In particular, we cannot have $\Psi(k) = \log\log\log k - \omega(1)$. □

4 Upper Bound

The lower bound on the advice size given in Corollary 3 is tight, as $O(\log\log k)$ bits of advice are sufficient to obtain an $O(1)$-competitive search algorithm. To further illustrate the power of Theorem 5, we now claim that the lower bound mentioned in Corollary 4 is also tight. Theorem 2 follows by combining the Theorem 6 below and Corollary 4. Due to space considerations, the proof of Theorem 6 is deferred to the full version of this paper.

Theorem 6. *There exists an $O(\log k)$-competitive algorithm using $\log\log\log k + O(1)$ bits of advice.*

5 Implications for Biology

A common problem, when studying a biological system is the complexity of the system and the huge number of parameters involved. This raises the need for more concise descriptions and several alternatives have been explored. One tactic is reducing the parameter space. This is done by dividing the parameter space into critical and non-critical directions where changes in non-critical parameters do not affect overall system behavior [19,30]. A different approach involves the definitions of bounds which govern biological systems independently of any algorithms or parameters. Previous works have utilized physics [7] and information theory [44] to set such bounds.

Our results are an attempt to draw non-trivial bounds on biological systems from the field of *distributed computing*. Such theoretical lower bounds on advice size may enable one to relate group search performance to the extent of information sharing within the nest. These types of bounds are particularly interesting since they provide not a single rule but relations between key parameters. In our case these would be the memory capacity of an agent and collective search efficiency.

We do not claim that our setting precisely captures the framework in which the aforementioned species perform search, yet we do believe that it provides a first approximation for it. Indeed, apart from engaging in central-place foraging with no mid-search communication, these species possess many of the individual skills required for the behavioral patterns that are utilized in our upper bounds [32,47,48,49,52,53]. It is not unreasonable to assume that a careful inspection of these species in nature would reveal a slightly different framework and would require the formulation of similar suitable theoretical memory bounds. Combining such memory lower bounds with experimental measurements of search speed with varying numbers of searchers would then provide quantitative evidence regarding the number of memory bits (or, alternatively, the number of states) used by ants during a search. In particular, this would help to understand the ants' quorum sensing process, as this number of memory bits are required merely for representing the output of that process. Although such experiments are beyond the scope of the current work, our results provide a "proof-of-concept" for this methodology.

References

1. Afek, Y., Alon, N., Barad, O., Hornstein, E., Barkai, N., Bar-Joseph, Z.: A biological solution to a fundamental distributed computing problem. Science 331(6014), 183–185 (2011)
2. Alpern, S., Gal, S.: The Theory of Search Games and Rendezvous, 319 p. Kluwer (now Springer) Academic Publishers (2003)
3. Albers, S., Henzinger, M.R.: Exploring unknown environments. SIAM J. on Computing 29, 1164–1188 (2000)
4. Averbakh, I., Berman, O.: $(p-1)/(p+1)$-approximate algorithms for p-traveling salesmen problems on a tree with minmax objective. Discr. Appl. Mathematics 75, 201–216 (1997)

5. Baeza-Yates, R.A., Culberson, J.C., Rawlins, G.J.E.: Searching in The Plane. Information and Computation 106(2), 234–252 (1991)
6. Berkolaiko, G., Havlin, S.: Territory covered by N Levy flights on d-dimensional lattices. Physical Review. E 55(2), 1395–1400 (1999)
7. Bialek, W.: Physical limits to sensation and perception. Annual Review of Biophysics and Biophysical Chemistry 16, 455–478 (1987)
8. Bonifaci, V., Mehlhorn, K., Varma, G.: Physarum can compute shortest paths. In: Proc. 23th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 233–240 (2012)
9. Le Breton, J., Fourcassié, V.: Information transfer during recruitment in the ant Lasius niger L (Hymenoptera: Formicidae). Behavioral Ecology and Sociobiology 55(3), 242–250 (2004)
10. Burroughs, N.J., de, M., de Oliveira, B.M.P.M., Adrego, P.A.: Regulatory Tcell adjustment of quorum growth thresholds and the control of local immune responses. J. of Theoretical Biology 241, 134–141 (2006)
11. Chazelle, B.: Natural algorithms. In: SODA 2009, pp. 422–431 (2009)
12. Cohen, R., Fraigniaud, P., Ilcinkas, D., Korman, A., Peleg, D.: Label-Guided Graph Exploration by a Finite Automation. ACM Transactions on Algorithms (TALG) 4(4) (2008)
13. Dessmark, A., Pelc, A.: Optimal Graph Exploration without Good Maps. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 374–386. Springer, Heidelberg (2002)
14. Diks, K., Fraigniaud, P., Kranakis, E., Pelc, A.: Tree exploration with little memory. In: SODA 2002, pp. 588–597 (2002)
15. Emek, Y., Fraigniaud, P., Korman, A., Rosen, A.: Online Computation with Advice. Theoretical Computer Science (TCS) 412(24), 2642–2656 (2011)
16. Feare, C.J., Dunnet, G.M., Patterson, I.J.: Ecologicalstudies of the rook (*Corvus frugilegus L.*) in north-east Scotland; Food intake and feeding behaviour. J. of Applied Ecology 11, 867–896 (1974)
17. Feinerman, O., Jentsch, G., Tkach, K.E., Coward, J.W., Hathorn, M.M., Sneddon, M.W., Emonet, T., Smith, K.A., Altan-Bonnet, G.: Single-cell quantification of IL-2 response by effector and regulatory T cells reveals critical plasticity in immune response. Molecular Systems Biology 6(437) (2010)
18. Feinerman, O., Korman, A., Lotker, Z., Sereni, J.S.: Collaborative Search on the Plane without Communication. To appear in PODC 2012 (2012)
19. Feinerman, O., Veiga, J., Dorfman, J.R., Germain, R.N., Altan-Bonnet, G.: Variability and robustness in T Cell activation from regulated heterogeneity in protein levels. Science 321(5892), 1081–1084 (2008)
20. Flocchini, P., Ilcinkas, D., Pelc, A., Santoro, N.: Remembering without memory: tree exploration by asynchronous oblivious robots. TCS 411, 1583–1598 (2010)
21. Flocchini, P., Ilcinkas, D., Pelc, A., Santoro, N.: How many oblivious robots can explore a line. Inf. Process. Lett. 111(20), 1027–1031 (2011)
22. Fraigniaud, P., Gąsieniec, L., Kowalski, D.R., Pelc, A.: Collective Tree Exploration. In: Farach-Colton, M. (ed.) LATIN 2004. LNCS, vol. 2976, pp. 141–151. Springer, Heidelberg (2004)
23. Fraigniaud, P., Ilcinkas, D., Pelc, A.: Oracle size: a new measure of difficulty for communication tasks. In: PODC 2006, pp. 179–187 (2006)
24. Fraigniaud, P., Gavoille, C., Ilcinkas, D., Pelc, A.: Distributed Computing with Advice: Information Sensitivity of Graph Coloring. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 231–242. Springer, Heidelberg (2007)

25. Fraigniaud, P., Korman, A., Lebhar, E.: Local MST Computation with Short Advice. Theory of Computing Systems (ToCS) 47(4), 920–933 (2010)
26. Fraigniaud, P., Korman, A., Peleg, D.: Local Distributed Decision. In: FOCS 2011 (2011)
27. Gasieniec, L., Pelc, A., Radzik, T., Zhang, X.: Tree exploration with logarithmic memory. In: SODA (2007)
28. Giraldeau, L.A., Carco, T.: Social Foraging Theory (2000)
29. Gordon, D.M.: The regulation of foraging activity in red harvester ant colonies. The American Naturalist 159(5), 509–518 (2002)
30. Gutenkunst, R.N., Waterfall, J.J., Casey, F.P., Brown, K.S., Myers, C.R., Sethna, J.P.: Universally sloppy parameter sensitivities in systems biology models. PLOS Computational Biology 3(10), e189 (2007), doi:10.1371/journal.pcbi.0030189
31. Hanusse, N., Kavvadias, D.J., Kranakis, E., Krizanc, D.: Memoryless search algorithms in a network with faulty advice. TCS 402(2-3), 190–198 (2008)
32. Harkness, R.D., Maroudas, N.G.: Central place foraging by an ant (Cataglyphis bicolor Fab.): a model of searching. Animal Behavior 33(3), 916–928 (1985)
33. Janeway, C.A., Travers, P., Walport, M., Shlomchik Immunobiology, M.J.: The Immune System in Health and Disease. Garland Science, New Yoy (2001)
34. Kao, M., Reif, J.H., Tate, S.R.: Searching in an Unknown Environment: An Optimal Randomized Algorithm for the Cow-Path Problem. J. of Inf. Comput., 63–79 (1996)
35. Korman, A., Sereni, J.S., Viennot, L.: Toward More Localized Local Algorithms: Removing Assumptions Concerning Global Knowledge. In: PODC 2011 (2011)
36. Korman, A., Kutten, S.: Distributed Verification of Minimum Spanning Trees. Distributed Computing (DC) 20(4) (2007)
37. Korman, A., Kutten, S., Peleg, D.: Proof Labeling Schemes. Distributed Computing (DC) 22(4) (2010)
38. Krebs, J.: Optimal foraging, predation risk and territory defense. Ardea 68, 83–90 (1980), Nederlandse Ornithlogische Unie
39. López-Ortiz, A., Sweet, G.: Parallel searching on a lattice. In: CCCG 2011, pp. 125–128 (2001)
40. Orians, G.F., Pearson, N.E.: On the theory of central place foraging. Analysis of Ecological Systems, 155–177 (1979)
41. Panaite, P., Pelc, A.: Exploring unknown undirected graphs. J. of Algorithms 33, 281–295 (1999)
42. Polycarpouy, M.M., Yang, Y., Passinoz, K.M.: A Cooperative Search Framework for Distributed Agents. In: Intelligent Control, pp. 1–6 (2001)
43. Pratt, S.C.: Quorum sensing by encounter rates in the ant Temnothorax albipennis. Behavioral Ecology 16(2), 488–496 (2005)
44. Rieke, F., Warland, D., Bialek, W.: Coding efficiency and information rates in sensory neurons. Europhysics Letters 22(2), 15–156 (1993)
45. Reingold, O.: Undirected connectivity in log-space. J. ACM 55(4) (2008)
46. Reynolds, A.M.: Cooperative random Lévy flight searches and the flight patterns of honeybees. Physics Letters A 354, 384–388 (2006)
47. Reynolds, A.M.: Optimal random Lévy-loop searching: New insights into the searching behaviours of central-place foragers. European Physics Letters 82(2), 20001 (2008)
48. Sommer, S., Wehner, R.: The ant's estimation of distance travelled: experiments with desert ants, Cataglyphis fortis. J. of Comparative Physiology A 190(1), 1–6 (2004)

49. Srinivasan, M.V., Zhang, S., Altwein, M., Tautz, J.: Honeybee Navigation: Nature and Calibration of the Odometer. Science 287, 851–853 (2000)
50. Surette, M.G., Miller, M.B., Bassler, B.L.: Quorum sensing in Escherichia coli, Salmonella typhimurium, and Vibrio harveyi: a new family of genes responsible for autoinducer production. Proc. National Acadamy of Science 96, 1639–1644 (1999)
51. Town, C.D., Gross, J.D., Kay, R.R.: Cell differentiation without morphogenesis in Dictyostelium discoideum. Nature 262, 717–719 (1976)
52. Wehner, R., Meier, C., Zollikofer, C.: The ontogeny of foraging behaviour in desert ants, Cataglyphis bicolor. Ecol. Entomol. 29, 240–250 (2004)
53. Wehner, R., Srinivasan, M.Y.: Searching behaviour of desert ants, genus Cataglyphis (Formicidae, Hymenoptera). J. of Comparative Physiology 142(3), 315–338 (1981)
54. Zahavi, A.: The function of pre-roost gatherings and communal roosts. Ibis 113, 106–109 (1971)

A Generalized Algorithm for Publish/Subscribe Overlay Design and Its Fast Implementation

Chen Chen[1], Roman Vitenberg[2], and Hans-Arno Jacobsen[1]

[1] Department of Electrical and Computer Engineering, University of Toronto, Canada
[2] Department of Informatics, University of Oslo, Norway
{chenchen,jacobsen}@eecg.toronto.edu, romanvi@ifi.uio.no

Abstract. It is a challenging and fundamental problem to construct the underlying overlay network to support efficient and scalable information distribution in topic-based publish/subscribe systems. Existing overlay design algorithms aim to minimize the node fan-out while building topic-connected overlays, in which all nodes interested in the same topic are organized in a directly connected dissemination sub-overlay. However, most state-of-the-art algorithms suffer from high computational complexity, such as $O(|V|^4|T|)$, where V is the node set and T is the topic set.

We devise a general indexing data structure that provides a significantly faster implementation, with $O(|V|^2|T|)$ running time, for different state-of-the-art algorithms. The generality of the indexing data structure is due to the fact that it enables edge lookup by both node degree and *edge contribution*, a central metric in all existing algorithms. When tested on typical pub/sub workloads, the speedup observed was by a factor of over 1 000, thereby rendering the algorithms more suitable for practical use. For example, under a typically Zipf distributed pub/sub workload, with 1 000 nodes and 100 topics, our new implementation completes in 3.823 seconds, while the previous alternative takes over 555 minutes.

1 Introduction

Publish/subscribe (pub/sub) systems constitute an attractive choice as communication paradigm and messaging substrate for building large-scale distributed systems. Many real-world applications are using pub/sub for message dissemination, such as application integration across data centers [27], financial data dissemination [3], RSS feed aggregation, filtering, and distribution [23,26], and business process management [21]. Google's GooPS [27] and Yahoo's YMB [15] constitute the distributed messaging substrates for online applications operating worldwide, TIBCO RV [3] has been used extensively for NASDAQ quote dissemination and order processing, and GDSN (Global Data Synchronization Network) [1] is a global pub/sub network enabling suppliers and retailers to exchange timely and accurate supply chain data.

In a distributed pub/sub system, so called pub/sub brokers, often connected in a federated manner as an application-level overlay network, efficiently route publication messages from data sources to sinks. The overlay of a pub/sub system directly impacts the system's performance and the message routing cost. Constructing a high-quality broker overlay is a key challenge and fundamental problem for distributed pub/sub systems that has received attention both in industry [27,15] and academia [13,24,25,18,7,10].

The notion of topic-connectivity is defined for topic-based pub/sub overlays [13], which informally speaking means that all nodes (i.e., pub/sub brokers) interested in the same topic are organized in a connected dissemination sub-overlay. This property ensures that nodes not interested in a topic would never need to contribute to disseminating information on that topic. Publication routing atop such overlays saves bandwidth and computational resources otherwise wasted on forwarding messages of no interest to the node. It also results in smaller routing tables. From a security perspective, topic-connectivity is desirable when messages are to be shared across a network among a set of trusted users without leaving this set.

Apart from topic-connectivity, it is imperative for an overlay network to have a low node degree. It costs a lot of resources to maintain adjacent links for a high-degree node (i.e., monitor the links and the neighbors [13,25]). Besides, for a typical pub/sub system, each link would have to accommodate a number of protocols, service components, message queues and so on. While overlay designs for different applications might be principally different, they all share the strive for maintaining bounded node degrees, whether in DHTs [22], wireless networks [16], or for survivable network design [19].

Several centralized algorithms have been proposed for constructing topic-connected overlays with the average node degree or the maximum node degree provably close to the optimal ones [13,24,25,10,12]. These state-of-the-art algorithms target overlay construction in a managed large cluster of up to thousands of servers where full mesh solutions exhibit scalability problems [27,15]. Such clusters are characterized by a large degree of control and relatively low churn rates (in the order of one change every hour, depending on the size of the cluster [2]), which makes centralized overlay construction a viable solution. Besides, these algorithms serve as stepping stones and comparison baselines for dynamic environments and decentralized overlay construction protocols.

However, the algorithms in [24,25,12] have the prohibitively expensive runtime cost of $O(|V|^4|T|)$ where $|V|$ is the number of nodes and $|T|$ is the number of topics. This fundamental drawback makes the algorithms non-suitable for the managed cluster environment because it takes tens of minutes or hours to compute an overlay for a realistic scale on a high-end machine. The runtime cost also limits the applicability of the algorithms as a comparison baseline.

The main contribution of this paper is that we generalize the above algorithms and come up with a new indexing data structure that supports a significantly faster implementation, with $O(|V|^2|T|)$ time efficiency. Specifically, all algorithms follow the same pattern: they iteratively add edges until the resulting overlay satisfies topic-connectivity. The data structure that we propose exhibits the following properties: (a) its initialization complexity is $O(|V|^2|T|)$, (b) the cumulative complexity of selecting an edge at all iterations is $O(|V|^2|T|)$, and (c) the amortized complexity of updating the data structure over all iterations is also $O(|V|^2|T|)$. The generality of the indexing data structure is due to the fact that it allows edge lookup by both node degree and the *edge contribution*, a central metric in the above algorithms.

To complement the theoretical analysis, we conduct comprehensive experiments under a variety of characteristic pub/sub workloads. Our experiments show that on average, for a typical pub/sub scale and interest distribution, our generalized algorithm with its efficient implementation builds the same overlay as previously known state-of-the-art algorithms in less than 0.37% of the running time. For example, under the Zipf distributed pub/sub workload, with 1000 nodes and 100 topics, our new implementation completes in 3.823 seconds, while the previous alternative takes over 555 minutes.

2 Related Work

The research in distributed pub/sub systems has been considering two main directions: (1) the design of routing protocols with emphasis on the efficiency and scalability of message dissemination from numerous publishers to a large number of subscribers (see for example: [28,8,20,4]) and (2) the construction of the underlying overlay topology such that network traffic is minimized (see for example: [13,24,18,7,10,25,17,12]). This paper focuses on the latter direction.

Topic-connectivity is a required property in [6,14]. It is also an implicit requirement in [8,5,7,9,17], which all aim to reduce the number of unnecessary intermediate overlay hops for message delivery using a variety of techniques.

Chockler *et al.* [13] introduced the parametrized family of *Scalable Overlay Construction (SOC)* design problems for pub/sub that captures the trade-off between the overlay scalability and the cost of message dissemination. They specifically focus on the MinAvg-TCO problem of minimizing the average node degree of the topic-connected overlay [13]. They proved the NP-Completeness of MinAvg-TCO and proposed the GM (Greedy Merge) algorithm that achieves a logarithmic approximation ratio with regard to average node degree [13]. Chen *et al.* [10] use GM as a building block for designing a divide-and-conquer approach to overlay design for pub/sub systems. This approach significantly reduces the time and space complexity of constructing a topic-connected overlay with a low average node degree.

Onus and Richa [24] analyzed the MinMax-TCO problem of minimizing the maximum degree of a topic-connected overlay network. They present the MinMax-ODA (Minimum Maximum Degree Overlay Design Algorithm) that attains a logarithmic approximation ratio on the maximum node degree. Chen *et al.* [12] focus on providing an efficient solution for MinMax-TCO by combining greedy and divide-and-conquer algorithm design techniques.

The GM and MinMax-ODA algorithms each focus on minimizing one single node degree metric, either average or maximum node degree. Each algorithm was shown to perform poorly with respect to the complementary metric. Onus and Richa [25] introduced the Low-TCO problem for minimizing both average and maximum node degrees in a topic-connected pub/sub overlay design at the same time. The authors designed the Low-ODA (Low Degree Overlay Design Algorithm), which achieves sub-linear approximations for both metrics [25].

Both MinMax-ODA and Low-ODA have the high time complexity of $O(|V|^4|T|)$, where $|V|$ is the number of nodes and $|T|$ is the number of topics. In this paper, we provide a generalization of the GM, MinMax-ODA, and Low-ODA algorithms and propose a fast implementation for the generalized algorithm with running time $O(|V|^2|T|)$. This speedup technique is also applicable to the algorithms proposed by Chen *et al.* [10,12] that can use the generalized algorithms as building blocks.

3 Background

In this section we present some definitions and background information essential for the understanding of the algorithms developed in this paper.

Let V be the set of nodes and T be the set of topics. The interest function Int is defined as $Int : V \times T \rightarrow \{true, false\}$. Since the domain of the interest function is a Cartesian product, we also refer to this function as an interest matrix. Given an interest function Int, we say that a node v is interested in some topic t if and only if $Int(v, t) = true$. We then also say that node v subscribes to topic t.

An overlay network $G(V, E)$ is an undirected graph over the node set V with the edge set $E \subseteq V \times V$. Given an overlay network $G(V, E)$, an interest function Int, and a topic $t \in T$, we say that a sub-graph $G_t(V_t, E_t)$ of G is *induced* by t if $V_t = \{v \in V | Int(v, t)\}$ and $E_t = \{(v, w) \in E | v \in V_t \wedge w \in V_t\}$. An overlay G is called *topic-connected* if for each topic $t \in T$, the sub-graph G_t of G induced by t contains at most one *topic-connected component* (*TC-component*). A *topic-connected overlay* (TCO) is denoted as $TCO(V, T, Int, E)$, TCO in short.

The concept of TCO is applicable to both P2P solutions for pub/sub in which the clients form the TCO and broker-based solutions in which the brokers form the TCO. It does not differentiate between publishers and subscribers. This abstraction simplifies the presentation for a theoretical and algorithmic treatment of the problem, while fully preserving its practical character. Aiming to achieve topic-connectivity while optimizing node degrees has resulted in the formulation of various problems: MinAvg-TCO for average degree [13], MinMax-TCO for maximum degree [24], and Low-TCO for both average degree and maximum degree simultaneously [25].

Problem 1. MinAvg-TCO(V, T, Int): Given a set of nodes V, a set of topics T, and the interest function Int, construct a topic-connected overlay which has the least possible total number of edges (i.e., the least possible average node degree).

Problem 2. MinMax-TCO(V, T, Int): Given a set of nodes V, a set of topics T, and the interest function Int, construct a topic-connected overlay with the smallest possible maximum node degree.

Problem 3. Low-TCO(V, T, Int): Given a set of nodes V, a set of topics T, and the interest function Int, construct a topic-connected overlay with both low average and low maximum node degree.

MinAvg-TCO and MinMax-TCO are proven NP-Complete [13,24]. Low-TCO integrates the optimization objectives of the former problems. Approximation algorithms are proposed for these TCO construction problems, all following a greedy heuristic: the GM algorithm for MinAvg-TCO [13], the MinMax-ODA algorithm for MinMax-TCO [24], and the Low-ODA algorithm for Low-TCO [25].

4 Generalized Overlay Design Algorithm

In this section, we introduce Gen-ODA (Generalized Overlay Design Algorithm) as specified in Alg. 1. It captures the similarities embedded in the GM, MinMax-ODA, and Low-ODA algorithms and offers an easy-to-specialize pattern for studying families of algorithms for solving TCO design problems. We illustrate some of the specializations of this pattern in this paper.

Gen-ODA starts with the overlay $G(V, E_{new})$ where $E_{new} = \emptyset$ so that there are $\left|\{v : Int(v,t)\}\right|$ singleton *TC-components* for each topic $t \in T$, i.e., there are $\sum_{t\in T} \left|\{v{:}Int(v,t)\}\right|$ separate *TC-components* in total. The algorithm progresses by adding edges to E_{new}, thus merging *TC-components* until $G(V, E_{new})$ contains at most one *TC-component* for each $t \in T$, i.e., the resulting overlay is topic-connected.

At each step, an edge e is selected from the potential edge set E_{pot} by **findEdge**() in Line 6 of Alg. 1. Specific algorithms for different TCO problems have their own rules for edge selection, i.e., **findEdge**() is a *virtual* function that needs to be overwritten with an implementation of a concrete criterion, which governs edge selection. We next illustrate these rules for the above listed algorithms. The rules are based on a combination of two criteria: node degree and *edge contribution*, which is defined as reduction in the number of *TC-components* caused by the addition of the edge to the current overlay. The edge contribution for e is denoted as $contrib(e)$.

Alg. 1. Generalized Overlay Design Algorithm

```
Gen-ODA($V, T, Int$)
Input: $V, T, Int$
Output: A topic-connected overlay $TCO(V, T, Int, E)$
1: $E_{new}, E_{pot} \leftarrow \emptyset$
2: for all $e{=}(v, w)$ s.t. $(w, v){\notin}E_{pot}$ where $v, w{\in}V$ do
3:     add $e$ to $E_{pot}$
4: initDataStructures()
5: while $G(V, E_{new})$ is not topic-connected do
6:     $e \leftarrow$ findEdge()
7:     $E_{new} \leftarrow E_{new} \cup \{e\}$
8:     $E_{pot} \leftarrow E_{pot} - \{e\}$
9:     updateDataStructures($e$)
10: return $TCO(V, T, Int, E_{new})$
```

1. Chockler *et al.* [13] use the **GM-rule** for edge selection with regard to MinAvg-TCO: GM greedily selects an edge with the highest contribution (regardless of the node degree). An optimized implementation of GM has the runtime of $O(|V|^2|T|)$. GM achieves a logarithmic approximation ratio for the average node degree; however, GM only provides an approximation ratio of $\Theta(|V|)$ for the maximum node degree [24].

2. Onus *et al.* [24] use the **MinMax-ODA-rule** for edge selection with regard to MinMax-TCO: MinMax-ODA also selects the edge with the highest contribution, but only among the edges that would minimally increase the maximum node degree. MinMax-ODA always produces a TCO that has a maximum node degree within at most $\log(|V||T|)$ times the optimal maximum node degree.However, MinMax-ODA only attains an approximation ratio of $\Theta(|V|)$ for the average node degree [25].

3. Onus *et al.* [25] propose the **Low-ODA-rule** for solving the Low-TCO problem: Low-ODA uses a parameter k to trade off the balance between average and maximum node degrees. The algorithm makes a weighed selection between the edge e_1 chosen by the **GM-rule** and the edge e_2 selected by the **MinMax-ODA-rule**: If $contrib(e_1)$ is greater than $k \cdot contrib(e_2)$, e_1 is added; otherwise e_2 is added. Low-ODA achieves sub-linear approximation ratios on both average and maximum node degrees.

Both MinMax-ODA and Low-ODA find an edge in $O(|V|^2)$ time by scanning all potential edges in a brute force manner, which leads to the time complexity of $O(|V|^4|T|)$ [24,25]. This runtime cost is the main impediment for deploying the algorithms in a relatively static cluster environment where the large degree of control makes a centralized overlay construction feasible. Furthermore, it limits the scale of validation for MinMax-ODA and Low-ODA which in turn diminishes the potential for using these algorithms as the building blocks (e.g., in the design of divide and conquer algorithms [12]) and the comparison baselines for distributed alternatives.

5 Fast Implementation of TCO Algorithms

This section offers an efficient implementation for our proposed Gen-ODA algorithm pattern and its various instantiations. With Alg. 1 as the common pattern, functions **initDataStructures()** and **updateDataStructures()** are shared by different instantiations of Gen-ODA, while **findEdge()** is specialized for different edge selection rules. The fast implementation is based on the new indexing structure that we introduce in this work. A simpler structure was used in [13], which only provided indexing by the edge contribution. In contrast, the structure we propose in this work allows for indexing both by the edge contribution and node degree. In particular, the use of this structure allows us to implement a faster version of MinMax-ODA and Low-ODA running in $O(|V|^2|T|)$ time. By using this faster version, we can accelerate the efficiency of divide-and-conquer algorithms proposed in [12].

We first present the central data structures and elementary functions utilized in our fast implementation of Gen-ODA. Then, we describe the implementation of functions **initDataStructures()** and **updateDataStructures()** shared by different instantiations. Finally, we show how to realize different edge selection rules under the umbrella of this common algorithm pattern. We prove results about the runtime complexity for each of these elements, which allows us to derive the total complexity of $O(|V|^2|T|)$. Table 1 summarizes the overlay construction problems and algorithms, which will be discussed in this section.

Table 1. Algorithms for Solving the TCO Problems

MinAvg-TCO	**Minimum Average Degree TCO Problem**
GM	Greedy Merge algorithm [13], $O(|V|^2|T|)$
F-MinAvg-ODA	Fast implementation for GM, $O(|V|^2|T|)$
MinMax-TCO	**Minimum Maximum Degree TCO Problem**
MinMax-ODA	Minimum Maximum Degree Overlay Design Algorithm [24], $O(|V|^4|T|)$
F-MinMax-ODA	Fast MinMax-ODA, $O(|V|^2|T|)$
Low-TCO	**Low Avg and Max Degree TCO Problem**
Low-ODA	Low Degree Overlay Design Algorithm [25], $O(|V|^4|T|)$
F-Low-ODA	Fast Low-ODA, $O(|V|^2|T|)$
TCO_{ALG}*	The TCO produced by ALG
$\mathbb{T}_{\text{ALG}}$	Running time of ALG

* where ALG stands for any of the discussed algorithms.

5.1 An Indexing Data Structure

We introduce an indexing data structure, $EdgeContrib$, as the underlying bedrock for our fast implementation of Gen-ODA. We opt to present $EdgeContrib$ in Class 2 using object-oriented design principles, because: (1) it provides a standard interface that can be reused efficiently to develop key functions of Gen-ODA; and (2) the grouping of data and procedures facilitates reasoning about the algorithms and time complexity.

$EdgeContrib$ defines an internal class EDGESTRUCT, which encapsulates an edge and meta-information about it, such as its contribution. Besides, EDGESTRUCT contains pointer fields $prev$ and $next$ to allow inclusion into a doubly-linked list.

EdgeContrib contains two additional member attributes: *edgeArray* and *edgeMap*. As illustrated in Fig. 1(a), member *edgeArray* is a 2-dimensional array of size $|T|\times|V|$, which is designed for quick search for the "best" edge at each iteration of Alg. 1. Each entry *edgeArray*$[c][d]$ contains a doubly-linked list of EDGESTRUCT objects corresponding to different edges with contribution c and higher node degree d.

Member *edgeMap* is a hashtable such that given an edge e, *edgeMap* allows for an efficient lookup of the corresponding EDGESTRUCT(e). In a well-dimensioned hashtable, arbitrary insertions, lookups and deletions have a constant average time cost per operation.

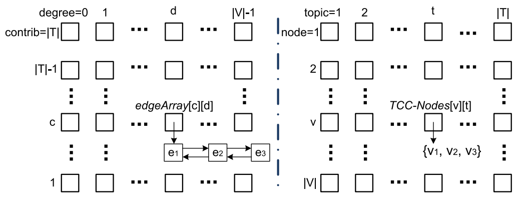


Fig. 1. (a) *EdgeContrib.edgeArray* (b) *TCC-Nodes*

Class 2. *EdgeContrib* Interface and Implementation

// Definition of EDGESTRUCT - data structure for *EdgeContrib* entries

EDGESTRUCT: an encapsulation of an edge and its corresponding information. It is implemented as an element in a doubly-linked list, so that inserting and deleting an edge can be performed in constant time.

- ∘ $e(v, w)$: the edge
- ∘ *contrib*: the edge contribution of $e(v, w)$, i.e., $contrib(e)$
- ∘ *degree*: $\max\{deg(v), deg(w)\}$ where $deg(v)$ is the degree of node v in $G(V, E_{new})$
- ∘ *prev*: pointer to its predecessor in the linked list
- ∘ *next*: pointer to its successor in the linked list

// Member attributes and auxiliary variables for *EdgeContrib*

▷ *edgeArray*: a 2-dimensional array with $|T|\times|V|$ entries, each representing a set of edges (and their corresponding information) chosen from $V\times V$. An edge $e(v, w)$ is wrapped in an EDGESTRUCT object (see the data structure definition above), denoted as EDGESTRUCT(e), when storing in an entry of *edgeArray*. If EDGESTRUCT$(e) \in$ *edgeArray*$[c][d]$, then: (1) $e \in E_{pot}$; (2) $c =$ EDGESTRUCT$(e).contrib$; (3) $d =$ EDGESTRUCT$(e).degree$.

▷ *edgeMap*: A hashtable that maps an edge e (as a key) to its associated EDGESTRUCT(e) (as a value) in *edgeArray*.

// Functions for *EdgeContrib*

```
▶ initEntry($c$, $d$)
1: $edgeArray[c][d] \leftarrow \emptyset$

▶ insertEdge($e(v, w)$, $c$, $d$)
1: construct EDGESTRUCT($e$) s.t. $contrib{=}c$ and $degree{=}d$
2: put key-value pair ($e$, EDGESTRUCT($e$)) into edgeMap
3: insert EDGESTRUCT($e$) into $edgeArray[c][d]$

▶ deleteEdge($e(v, w)$, $c$, $d$)
1: get EDGESTRUCT($e$) from edgeMap using $e$ as the key
2: delete key-value pair ($e$, EDGESTRUCT($e$)) from edgeMap
3: delete EDGESTRUCT($e$) from $edgeArray[c][d]$

▶ getOneEdge($c$, $d$)
1: return the first edge from $edgeArray[c][d]$

▶ getContrib($e(v, w)$)
1: get EDGESTRUCT($e$) from edgeMap by key $e$
2: return EDGESTRUCT($e$).$contrib$

▶ getDegree($e(v, w)$)
1: get EDGESTRUCT($e$) from edgeMap by key $e$
2: return EDGESTRUCT($e$).$degree$

▶ entrySize($c$, $d$)
1: return $|edgeArray[c][d]|$
```

While the implementation of individual functions in *EdgeContrib* is rather straightforward, it is important to observe that each function has a per-invocation runtime cost of $O(1)$. Edge addition or deletion takes constant time thanks to the use of a doubly-linked list. Edge lookup takes $O(1)$ due to using the *edgeMap* hashtable. This property of the constant per-invocation cost is essential for the time efficiency of updating all EDGESTRUCTS in *edgeArray* after adding each edge to the overlay, as we further elaborate upon in Lemma 3.

5.2 A Common Template for Implementations

We have showed the outline of Gen-ODA in Alg. 1. A more detailed description with actual data structures for Gen-ODA is presented in the following algorithms: definitions of data structures (Alg. 3), initialization of data structures (Alg. 4) and the update of data structures after each edge addition (Alg. 5). GM [13], MinMax-ODA [24] and Low-ODA [25] all fit into the framework of the Gen-ODA, and the only difference is that they use different criteria to select an edge at each iteration (Line 6 of Alg. 1).

Alg. 3. Global Variables

- ▶ $EdgeContrib$: an indexing data structure designed for quick search for the best candidate edge using various edge selection rules. See Class 2.
- ▶ $TCC\text{-}Nodes$: a 2-dimensional array of size $|V|\times|T|$ in which each element $TCC\text{-}Nodes[v][t]$ is a subset of V s.t. for each $w \in TCC\text{-}Nodes[v][t]$, (1) $Int(w,t) = true$, and (2) both w and v belong to the same *TC-component* for t.
- ▶ E_{new}: set of edges in the overlay built so far.
- ▶ E_{pot}: set of potential edges that can be added.
- ▶ $nodeDegree$: an array with length $|V|$ s.t. $nodeDegree[v]$ is the degree of node v in $G(V, E_{new})$.
- ▶ $maxContrib$: the highest edge contribution in E_{pot}.
- ▶ $maxDegree$: the maximum node degree in $G(V, E_{new})$.
- ▶ $curContrib$: contribution of the currently selected edge.
- ▶ $curDegree$: the higher node degree of the currently selected edge.

Our implementation of Gen-ODA uses several global variables defined in Alg. 3. Among these data structures, $EdgeContrib$ and $TCC\text{-}Nodes$ play the most important roles (see Fig. 1). $EdgeContrib$ is an indexing data structure designed to organize all potential edges (see Class 2). $TCC\text{-}Nodes$ is a 2-dimensional array of size $|V|\times|T|$ which keeps track of the *TC-components* in the current overlay $G(V, E_{new})$: $TCC\text{-}Nodes[v][t]$ holds the set of nodes belonging to the same *TC-component* for t as v. To support all these variables for Gen-ODA, a polynomial space is sufficient.

Alg. 4. Data Structure Initialization

```
initDataStructures()
 1: for all $v \in V$ do
 2:     $nodeDegree[v] \leftarrow 0$
 3: for all $v \in V \wedge t \in T$ such that $Int(v,t)$ do
 4:     $TCC\text{-}Nodes[v][t] \leftarrow \{v\}$
 5: for $c \leftarrow |T|$ down to 1 do
 6:     for $d \leftarrow 0$ to $|V|-1$ do
 7:         $EdgeContrib.\mathtt{initEntry}(c,d)$
 8: for all $e = (v,w) \in E_{pot}$ do
 9:     $c \leftarrow |\{t \in T | Int(v,t) \wedge Int(w,t)\}|$
10:     if $c > 0$ then
11:         $EdgeContrib.\mathtt{insertEdge}(e,c,0)$
12: $maxContrib$
      $\leftarrow \max\{c \mid \exists\, d$ s.t. $EdgeContrib.\mathtt{entrySize}(c,d)>0\}$
13: $curContrib \leftarrow maxContrib$
14: $curDegree \leftarrow 0$, $maxDegree \leftarrow 0$
```

Lemma 1. *Alg. 1 takes $O(|V|^2|T|)$ space.*

The initialization of these data structures (Alg. 4) takes place at the very beginning of the Gen-ODA algorithm. Gen-ODA starts with the overlay $G(V, \emptyset)$, and Alg. 4 initializes all global variables defined in Alg. 3 accordingly. Lemma 2 shows the time complexity of the initialization.

Lemma 2. *The running time of Alg. 4 is $O(|V|^2|T|)$.*

Proof. The cost of Gen-ODA's initialization is dominated by the calculation of edge contribution for all potential edges E_{pot} in Lines 8-11 of Alg. 4. If the interest of each node is stored as a list of topics, then the complexity of this computation for E_{pot} will be $O(\sum_{e=(v,w)\in E_{pot}} |\{t \in T | Int(v,t) \wedge Int(w,t)\}|) = O(|V|^2|T|)$. □

After adding e to the overlay and removing it from the potential set (Line 7-8 in Alg. 1), nodes and edges ought to be re-arranged in *EdgeContrib* and *TCC-Nodes* dynamically to reflect the new edge contributions, *TC-components*, and node degrees (Line 9 in Alg. 1). This is performed by Alg. 5.

Alg. 5. Data Structure Update

```
updateDataStructures($e(v, w)$)
  // (1) Update variables for current edge
1:  $curContrib \leftarrow EdgeContrib$.getContrib($e$)
2:  $curDegree \leftarrow EdgeContrib$.getDegree($e$)
3:  $EdgeContrib$.deleteEdge($e$, $curContrib$, $curDegree$)
  // (2) Update contributions and TC-components
4:  for all $t \in T$ such that $Int(v,t) \wedge Int(w,t) \wedge TCC\text{-}Nodes[v][t] \neq TCC\text{-}Nodes[w][t]$ do
5:      for all $v' \in TCC\text{-}Nodes[v][t] \wedge w' \in TCC\text{-}Nodes[w][t] \wedge e'(v', w') \neq e(v, w)$ do
6:          $c \leftarrow EdgeContrib$.getContrib($e'$), $d \leftarrow EdgeContrib$.getDegree($e'$)
7:          $EdgeContrib$.deleteEdge($e'$, $c$, $d$)
8:          if $c > 1$ then
9:              $EdgeContrib$.insertEdge($e'$,$c-1$, $d$)
10:         else
11:             delete $e'$ from $E_{pot}$
12:     $new\_tcc\_nodes \leftarrow TCC\text{-}Nodes[v][t] \cup TCC\text{-}Nodes[w][t]$
13:     for all $u \in new\_tcc\_nodes$ do
14:         $TCC\text{-}Nodes[u][t] \leftarrow new\_tcc\_nodes$
  // (3) Update node degrees
15: $nodeDegree[v] \leftarrow nodeDegree[v] + 1$, $nodeDegree[w] \leftarrow nodeDegree[w] + 1$
16: $maxDegree \leftarrow$ max$\{maxDegree, nodeDegree[v], nodeDegree[w]\}$
17: for all $(v', w') \in E_{pot}$ that is incident on $v$ or $w$ do
18:     $d_{old} \leftarrow EdgeContrib$.getDegree($(v', w')$)
19:     $d_{new} \leftarrow$ max$\{nodeDegree[v'], nodeDegree[w']\}$
20:     if $d_{old} < d_{new}$ then
21:         $c \leftarrow EdgeContrib$.getContrib($(v', w')$)
22:         $EdgeContrib$.deleteEdge($(v', w')$, $c$, $d_{old}$)
23:         $EdgeContrib$.insertEdge($(v', w')$, $c$, $d_{new}$)
  // (4) Update $maxContrib$
24: while $maxContrib > 0$ do
25:     for $degree \leftarrow 0$ to $maxDegree$ do
26:         if $EdgeContrib$.entrySize($maxContrib$, $degree$) $> 0$ then
27:             break from while loop in Line 24
28:     $maxContrib \leftarrow maxContrib - 1$
```

As shown In Alg. 5, the update has four parts: (1) Lines 1-2 update $curContrib$ and $curDegree$ using the currently selected edge; (2) Lines 4-14 update edge contributions for EDGESTRUCTS stored in *EdgeContrib* and *TC-components* recorded in *TCC-Nodes*; (3) Lines 17-23 update the array entries in *EdgeContrib* according to node degrees of $G(V, E_{new})$; (4) Lines 24-28 update the global variable $maxContrib$.

Part (1) and Part (4) deal with basic data types, and are relatively straightforward. Parts (2) and Part (3) are responsible for handling complex data structures.

In Part (2), Lines 6-11 update the contribution of each edge affected by the addition of $e(v, w)$ to the overlay. An edge is affected if its endpoints belong to different *TC-components* prior to the addition but those components are merged as a result of the addition. Once edge $e(v, w)$ is added to the overlay, two *TC-components* are merged into a single one $new\_tcc\_nodes = TCC\text{-}Nodes[v][t] \cup TCC\text{-}Nodes[w][t]$ (Lines 12). Accordingly, for each node $u \in new\_tcc\_nodes$, $TCC\text{-}Nodes[u][t]$ is updated (Lines 13-14).

In Part (3), Alg. 5 handles the node degree update. Lines 15-16 update global variables $nodeDegree$ and $maxDegree$ following the addition of a new edge $e(v, w)$. Lines 17-23 examine all potential edges incident on either v or w and update the corresponding node degrees as the dimension in *EdgeContrib*.edgeArray. For each edge $e'(v', w')$, Line 18 retrieves the *old* degree as the index in *EdgeContrib*.edgeArray, and Line 19 computes the *new* degree in $G(V, E_{new})$; Lines 20-23 update the indexing structure if $d_{old} < d_{new}$.

Lemma 3 shows the cumulative running time of updates performed by Alg. 5 for all edges added to the TCO. We provide the detailed proof in our technical report [11].

Lemma 3. *The cumulative running time of all invocations of Alg. 5 during the entire execution of Alg. 1 is* $O(|V|^2|T|)$.

Having presented an efficient implementation of **initDataStructures**() and **updateDataStructures**() for Alg. 1, we now focus on the concrete realizations of **findEdge**() for different TCO construction criteria in Sec. 5.3, 5.4, and 5.5. At each iteration of Gen-ODA, **findEdge**() (Line 6 in Alg. 1) finds an edge e, whose addition would merge at least two different *TC-components* (for at least one topic), thus reducing the total number of *TC-components* by at least one. While naive search for the next "best" edge takes $O(|V|^2)$ time, the implementation presented here improves the time complexity by employing the auxiliary indexing data structure $EdgeContrib$. This data structure facilitates finding the 'best' edge at each iteration taking both edge contribution and node degree into account because the algorithm can traverse $EdgeContrib.\mathtt{edgeArray}[c][d]$ in the order of decreasing contribution c and increasing degree d and pick an edge from the first non-empty entry.

5.3 Finding Edge for MinAvg-TCO

Gen-ODA together with Alg. 6, referred to as F-MinAvg-ODA (Fast MinAvg Overlay Design Algorithm), builds the same overlay as GM [13]. Alg. 6 implements the **GM-rule**: it always chooses the edge with the highest contribution toward topic-connectivity regardless of node degrees.

Alg. 6. Find a MinAvg Edge

findMinAvgEdge()

Output: an edge e to be added to E_{new}

```
1: for $degree \leftarrow curDegree$ to $maxDegree$ do
2:     if $EdgeContrib.\mathtt{entrySize}(maxContrib, degree) > 0$ then
3:         $e \leftarrow EdgeContrib.\mathtt{getOneEdge}(maxContrib, degree)$
4:         return $e$
```

Lemma 4 shows that F-MinAvg-ODA achieves the same time efficiency as GM. The formal proof for Lemma 4 is omitted here, since it basically is a simplification of the time efficiency proof for F-MinMax-ODA, which we present in Sec. 5.4.

Lemma 4. *The cumulative running time of all invocations of Alg. 6 during the entire execution of Alg. 1 is* $O(|V|^2|T|)$.

5.4 Finding Edge for MinMax-TCO

MinMax-ODA in [24] yields the time complexity of $O(|V|^4|T|)$. Gen-ODA with the **MinMax-ODA-rule** implemented in Alg. 7 provides an efficient realization of MinMax-ODA, with an improved running time of $O(|V|^2|T|)$. We refer to this combined algorithm as F-MinMax-ODA (Fast MinMax-ODA).

In order to explain Alg. 7, we first observe that MinMax-ODA (and consequently F-MinMax-ODA) adds new edges in phases. At the start of each phase, MinMax-ODA selects a new edge that increases the maximum degree of the overlay by one. Then, the algorithm proceeds with adding edges without raising the maximum degree until the addition of any extra edge would cause a new increase, at which point the phase ends. The number of such phases is limited by the highest possible overlay degree, i.e., $O(|V|)$.

When invoked by Alg. 1 at each iteration, Alg. 7 scans the entries corresponding to non-maximum degree ($< maxDegree$) in edgeArray of $EdgeContrib$ in the order of increasing degree and decreasing contribution. If a non-empty entry is found, an arbitrary edge from the entry edge list is selected. Otherwise, an edge from the entry with the maximum contribution and maximum degree is selected, which leads to the increase in the overlay degree and signifies a start of a new phase.

Alg. 7. Find a MinMax Edge

```
findMinMaxEdge()
Output: an edge $e$ to be added to $E_{new}$
 1: $e \leftarrow$ NIL, $contrib \leftarrow curContrib$
 2: while $e =$ NIL $\wedge$ $contrib > 0$ do
 3:     $initDegree \leftarrow 0$
 4:     if $contrib = curContrib$ then
 5:         $initDegree \leftarrow curDegree$
 6:     for $degree \leftarrow initDegree$ to $maxDegree - 1$ do
 7:         if $EdgeContrib$.entrySize($contrib$, $degree$) $> 0$ then
 8:             $e \leftarrow EdgeContrib$.getOneEdge($contrib$, $degree$)
 9:             break from for loop in Line 6
10:     $contrib \leftarrow contrib - 1$
11: if $e =$ NIL then
12:     $e \leftarrow EdgeContrib$.getOneEdge($maxContrib$, $maxDegree$)
13: return $e$
```

The crucial element for the efficiency of the implementation is that rather than scanning the entire edgeArray of $EdgeContrib$ at each invocation, Alg. 7 continues the scan from the last selected entry. First, it does not affect the correctness of the scan: while after an edge addition, Alg. 5 reshuffles potential edges across edgeArray, it only moves the edges in the order of decreasing *contrib* (Lines 7-9) or increasing *degree* (Lines 22-23). Since Alg. 7 scans the entries in precisely the same order, it cannot miss a potential edge.

Secondly, continuing the scan from the last selected entry upon each Alg. 7 invocation within a single phase implies that the number of entries scanned at each phase is limited by the sum of two factors: the total number of entries in edgeArray of $EdgeContrib$ (which is equal to $|V| \cdot |T|$) plus the number of entries scanned multiple times, i.e., the number of Alg. 7 invocations, which is equal to the number of edges selected at this phase (which is limited by $\frac{|V|}{2}$ [24]). Therefore, the number of entries scanned during the entire execution of Alg. 1 (i.e., at all $O(|V|)$ phases) is $O(|V| \cdot (|V||T| + \frac{|V|}{2})) = O(|V|^2|T|)$. This underlines the proof of Lemma 5; complete proof can be found in our technical report [11].

Lemma 5. *The cumulative running time for all invocations of Alg. 7 during the entire execution of Alg. 1 is $O(|V|^2|T|)$.*

5.5 Finding Edge for Low-TCO

A naive implementation of Low-ODA yields the time complexity of $O(|V|^4|T|)$ (see Lemma 3 in [25]). The Gen-ODA implementing the **Low-ODA-rule** is described in Alg. 8, which we refer to as F-Low-ODA (Fast Low-ODA), produces the same overlay with the improved running time of $O(|V|^2|T|)$. Combined, Lemma 4 and Lemma 5 allow us to establish Lemma 6.

Alg. 8. Find a Low Edge

```
findLowEdge($k$)
Input: $k$: parameter to balance edge selection rules
Output: an edge $e$ to be added to $E_{new}$
1: $e_1 \leftarrow$ findMinAvgEdge(),
   $contrib_1 \leftarrow EdgeContrib$.getContrib($e_1$)
2: $e_2 \leftarrow$ findMinMaxEdge(),
   $contrib_2 \leftarrow EdgeContrib$.getContrib($e_2$)
3: if $contrib_1 \geq contrib_2 \times k$ then
4:     return $e_1$
5: else
6:     return $e_2$
```

Lemma 6. *The cumulative running time for all invocations of Alg. 8 during the entire execution of Alg. 1 is $O(|V|^2|T|)$.*

5.6 Running Time for Gen-ODA

To summarize all complexity analyses based on Lemmas 2, 3, 4, 5 and 6, the following lemma establishes the time efficiency of our implementation for F-MinAvg-ODA, F-MinMax-ODA and F-Low-ODA.

Lemma 7. *The running time of Alg. 1 with function **findEdge()** instantiated as either Alg. 6, Alg. 7 or Alg. 8 is $O(|V|^2|T|)$.*

6 Evaluation

We implement all algorithms in Table 1 in Java and evaluate the running time of different algorithms, i.e., F-MinMax-ODA (vs. MinMax-ODA) and F-Low-ODA (vs. Low-ODA). We denote by T_v the topic set which node v subscribes to, and by $|T_v|$ the *subscription size* of node v. In these experiments, we use the following value ranges as input: $|V| \in [100, 1\,000]$, $|T| \in [100, 1\,000]$, and $|T_v| \in [10, 100]$, where the subscription size is fixed for each node in the input. Each topic $t_i \in T$ is associated with probability q_i, $\sum_i q_i = 1$, so that each node subscribes to t_i with a probability q_i. The value of q_i is distributed according to either a uniform, a Zipf (with $\alpha = 2.0$), or an exponential distribution. According to [14], these distributions are representative of actual workloads used in industrial pub/sub systems today. Liu *et al.* [23] show that the Zipf distribution faithfully describes the feed popularity distribution in RSS feeds (a pub/sub-like application scenario). The exponential distribution is used by stock-market monitoring engines for the study of stock popularity in the New York Stock Exchange [29].

6.1 F-MinMax-ODA for MinMax-TCO

We now consider F-MinMax-ODA's performance compared to MinMax-ODA with respect to different input parameters. Both F-MinMax-ODA and MinMax-ODA algorithms use the **MinMax-ODA-rule** for edge selection but are based on different implementations. Since the TCOs they compute are the same, we only show their running time ratios here.

Fig. 2(a) depicts the comparison between F-MinMax-ODA and MinMax-ODA as the number of nodes increases when $|T| = 100$. As the figure shows, F-MinMax-ODA runs considerably faster. Under uniform distribution, $\mathbb{T}_{\mathsf{FMM}}$ is on average 0.858% of $\mathbb{T}_{\mathsf{MM}}$; under Zipf distribution, $\mathbb{T}_{\mathsf{FMM}}$ is on average 1.17% of $\mathbb{T}_{\mathsf{MM}}$. Additionally, the F-MinMax-ODA algorithm gains more speedup with the increase in the number of nodes compared to MinMax-ODA: when $|V|=1000$, $\mathbb{T}_{\mathsf{FMM}} = 0.0158\% \cdot \mathbb{T}_{\mathsf{MM}}$ for the uniform distribution and $\mathbb{T}_{\mathsf{FMM}} = 0.0115\% \cdot \mathbb{T}_{\mathsf{MM}}$ for the Zipf distribution. The gap in the running time between our algorithms and existing ones is so significant that instead of showing the absolute values on the same scale we opt to present the ratio. For example, under the Zipf distribution, with 1000 nodes and 100 topics, F-MinMax-ODA completes in 3.823 seconds, while MinMax-ODA takes over 555 minutes. This shows that F-MinMax-ODA provides an adequate solution for the above target settings while MinMax-ODA does not.

Fig. 2(b) depicts how F-MinMax-ODA and MinMax-ODA perform when the number of topics varies. The running time ratio of F-MinMax-ODA to MinMax-ODA increases as the number of topics increases from 100 to 1000. In order to explain this effect, we observe that the running time of scanning the indexing structure in F-MinMax-ODA is proportional to the maximum edge contribution while the running time of MinMax-ODA is independent of edge contributions. Increasing the number of topics leads to reduced correlation, i.e., the probability of having two nodes interested in the same topic drops as the number of topics increases, and with reduced correlation the edge contribution tends to be lower. This reduction in correlation is more pronounced for the uniform distribution of interests compared to skewed ones, such as Zipf. Yet, the increase in the running time ratio is not very significant: on average, F-MinMax-ODA is less than 0.236% of MinMax-ODA under the uniform distribution, and less than 0.019% under the Zipf distribution.

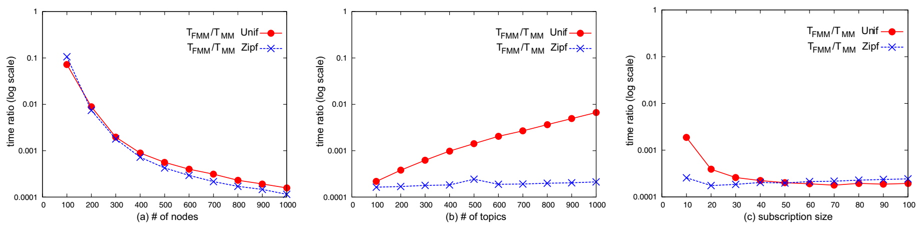


Fig. 2. F-MinMax-ODA vs. MinMax-ODA

Fig. 2(c) depicts the impacts of the subscription size on F-MinMax-ODA and MinMax-ODA. We set $|T| = 200$, and $|T_v|$ varies from 10 to 100. As shown in the figure, the ratio of $\mathbb{T}_{\text{FMM}}$ to $\mathbb{T}_{\text{MM}}$ decreases with the increase of $|T_v|$, and the ratio becomes relatively stable around 0.02% when $|T_v| > 50$.

6.2 F-Low-ODA for Low-TCO

We now explore the impact of different input variables on the performance of the F-Low-ODA and Low-ODA algorithms. Both apply the **Low-ODA-rule** for edge selection, so for the evaluation, we only consider their implementation efficiency.

Fig. 3(a) depicts the comparison between these two algorithms as the number of nodes increases where $|T|$=100. As the figure shows, F-Low-ODA runs significantly faster. Under the uniform distribution, $\mathbb{T}_{\text{FLOW}}$ is on average 1.2% of $\mathbb{T}_{\text{LOW}}$. Under the Zipf distribution, $\mathbb{T}_{\text{FLOW}}$ is on average 0.6% of $\mathbb{T}_{\text{LOW}}$. Additionally, F-Low-ODA gains more speedup with the increase in the number of nodes compared to Low-ODA: when $|V| = 1000$, $\mathbb{T}_{\text{FLOW}} = 0.15\% \cdot \mathbb{T}_{\text{LOW}}$ for the uniform distribution and $\mathbb{T}_{\text{FMM}} = 0.11\% \cdot \mathbb{T}_{\text{MM}}$ for the Zipf distribution.

Fig. 3(b) depicts the performance of F-Low-ODA and Low-ODA when we vary the number of topics. The ratio of $\mathbb{T}_{\text{FLOW}}$ to $\mathbb{T}_{\text{LOW}}$ increases as the number of topics increases from 100 to 1000, yet this effect is insignificant: on average, F-Low-ODA

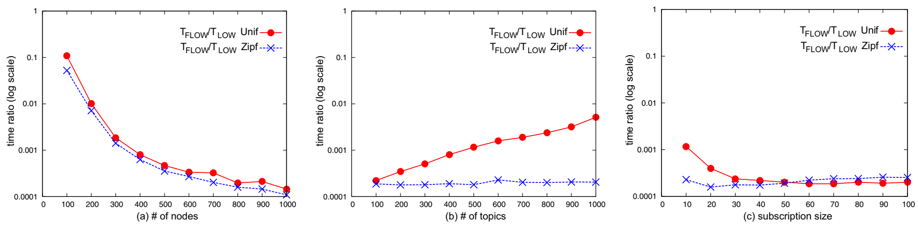


Fig. 3. F-Low-ODA vs. Low-ODA

takes less than 0.172% of Low-ODA's running time under the uniform distribution and less than 0.020% under the Zipf distribution. Further, F-Low-ODA has more speedup on the time efficiency for skewed distributions as the number of topics increases. The reason is that increasing the number of topics leads to less correlation, and under skewed distribution, the correlation among nodes drops relatively slower compared to that under the uniform distribution.

Fig. 3(c) depicts the effects of the subscription size on F-Low-ODA and Low-ODA. We set $|T|{=}200$ and $|T_v|{\in}[10, 100]$. As shown in the figure, the running time ratio $\frac{\mathbb{T}_{\mathsf{FLOW}}}{\mathbb{T}_{\mathsf{LOW}}}$ decreases with the increase of $|T_v|$. The ratio becomes stable around 0.02% as $|T_v|{>}50$.

7 Conclusions

In this paper, we develop the Gen-ODA framework that covers existing greedy algorithms with different edge selection rules for different optimization criteria. By using the indexing data structures that we have devised, a number of known algorithms gain a significant running time speedup, i.e., the time complexity of MinMax-ODA and Low-ODA is improved from $O(|V|^4|T|)$ to $O(|V|^2|T|)$.

We have evaluated the algorithms through a comprehensive experimental analysis, which demonstrates their performance and scalability under various practical pub/sub workloads. Our proposed Gen-ODA is well suited to different TCO construction problems: its efficient implementation accelerates the time efficiency by a factor of more than 1 000, and it gains more impact in the running time when the workloads scale up.

References

1. GDSN, `http://bit.ly/cjnevk`
2. Google Cluster Data, `http://code.google.com/p/googleclusterdata/`
3. TIBCO Rendezvous, `http://www.tibco.com`
4. Araujo, F., Rodrigues, L., Carvalho, N.: Scalable QoS-based event routing in publish-subscribe systems. In: NCA 2005 (2005)
5. Baehni, E., Eugster, P., Guerraoui, R.: Data-aware multicast. In: DSN 2004 (2004)
6. Baldoni, R., Beraldi, R., Quema, V., Querzoni, L., Tucci-Piergiovanni, S.: TERA: topic-based event routing for peer-to-peer architectures. In: DEBS 2007 (2007)
7. Baldoni, R., Beraldi, R., Querzoni, L., Virgillito, A.: Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA. Comput. J. 50(4) (2007)

8. Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.: SCRIBE: A large-scale and decentralized application-level multicast infrastructure. JSAC (2002)
9. Chand, R., Felber, P.: Semantic peer-to-peer overlays for publish/subscribe networks. In: EUROPAR 2005 (2005)
10. Chen, C., Jacobsen, H.-A., Vitenberg, R.: Divide and conquer algorithms for publish/subscribe overlay design. In: ICDCS 2010 (2010)
11. Chen, C., Vitenberg, R., Jacobsen, H.-A.: A generalized algorithm for publish/subscribe overlay design and its fast implementation. Tech. rep., U. of Toronto & U. of Oslo, http://msrg.org/papers/TRCVJ-GenODA
12. Chen, C., Vitenberg, R., Jacobsen, H.-A.: Scaling construction of low fan-out overlays for topic-based publish/subscribe systems. In: ICDCS 2010 (2010)
13. Chockler, G., Melamed, R., Tock, Y., Vitenberg, R.: Constructing scalable overlays for pub-sub with many topics: Problems, algorithms, and evaluation. In: PODC 2007 (2007)
14. Chockler, G., Melamed, R., Tock, Y., Vitenberg, R.: Spidercast: A scalable interest-aware overlay for topic-based pub/sub communication. In: DEBS 2007 (2007)
15. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. (2008)
16. De Santis, E., Grandoni, F., Panconesi, A.: Fast Low Degree Connectivity of Ad-Hoc Networks Via Percolation. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 206–217. Springer, Heidelberg (2007)
17. Girdzijauskas, S., Chockler, G., Vigfusson, Y., Tock, Y., Melamed, R.: Magnet: practical subscription clustering for internet-scale publish/subscribe. In: DEBS 2010 (2010)
18. Jaeger, M.A., Parzyjegla, H., Mühl, G., Herrmann, K.: Self-organizing broker topologies for publish/subscribe systems. In: SAC 2007 (2007)
19. Lau, L.C., Naor, J.S., Salavatipour, M.R., Singh, M.: Survivable network design with degree or order constraints. In: Proc. ACM STOC 2007 (2007)
20. Li, G., Muthusamy, V., Jacobsen, H.-A.: Adaptive Content-Based Routing in General Overlay Topologies. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 1–21. Springer, Heidelberg (2008)
21. Li, G., Muthusamy, V., Jacobsen, H.-A.: A distributed service oriented architecture for business process execution. In: ACM TWEB (2010)
22. Liben-Nowell, D., Balakrishnan, H., Karger, D.: Analysis of the evolution of peer-to-peer systems. In: PODC 2002 (2002)
23. Liu, H., Ramasubramanian, V., Sirer, E.G.: Client behavior and feed characteristics of RSS, a publish-subscribe system for web micronews. In: IMC 2005 (2005)
24. Onus, M., Richa, A.W.: Minimum maximum degree publish-subscribe overlay network design. In: INFOCOM 2009 (2009)
25. Onus, M., Richa, A.W.: Parameterized maximum and average degree approximation in topic-based publish-subscribe overlay network design. In: ICDCS 2010 (2010)
26. Petrovic, M., Liu, H., Jacobsen, H.-A.: G-ToPSS: fast filtering of graph-based metadata. In: WWW 2005 (2005)
27. Reumann, J.: Pub/Sub at Google. Lecture & Personal Communications at EuroSys & CANOE Summer School, Oslo, Norway (August 2009)
28. Tam, D., Azimi, R., Jacobsen, H.-A.: Building content-based publish/subscribe systems with distributed hash tables. In: DBISP2P 2003 (2003)
29. Tock, Y., Naaman, N., Harpaz, A., Gershinsky, G.: Hierarchical clustering of message flows in a multicast data dissemination system. In: IASTED PDCS (2005)

Bounded-Contention Coding for Wireless Networks in the High SNR Regime

Keren Censor-Hillel[1], Bernhard Haeupler[1], Nancy Lynch[1], and Muriel Médard[2]

[1] CSAIL, Massachusetts Institute of Technology, MA 01239, USA
[2] RLE, Massachusetts Institute of Technology, MA 01239, USA

Abstract. Efficient communication in wireless networks is typically challenged by the possibility of interference among several transmitting nodes. Much important research has been invested in decreasing the number of collisions in order to obtain faster algorithms for communication in such networks.

This paper proposes a novel approach for wireless communication, which embraces collisions rather than avoiding them, over an additive channel. It introduces a coding technique called *Bounded-Contention Coding (BCC)* that allows collisions to be successfully decoded by the receiving nodes into the original transmissions and whose complexity depends on a bound on the contention among the transmitters.

BCC enables *deterministic* local broadcast in a network with n nodes and at most a transmitters with information of ℓ bits each within $O(a \log n + a\ell)$ bits of communication with full-duplex radios, and $O((a \log n + a\ell)(\log n))$ bits, with high probability, with half-duplex radios. When combined with random linear network coding, BCC gives *global* broadcast within $O((D + a + \log n)(a \log n + \ell))$ bits, with high probability. This also holds in dynamic networks that can change arbitrarily over time by a worst-case adversary. When no bound on the contention is given, it is shown how to probabilistically estimate it and obtain global broadcast that is adaptive to the true contention in the network.

Keywords: wireless networks, high SNR, coding, additive channel.

1 Introduction

Handling interference in wireless networks is a fundamental challenge in designing algorithms for efficient communication. When two devices that are near each other transmit at the same time the result is a collided signal. In order to enable the receivers to obtain the original information that was sent, thereby achieving efficient communication that allows the design of fast algorithms for wireless networks, much important research has been invested in scheduling the transmissions in a way that avoids collisions as much as possible.

Avoiding collisions basically requires some type of symmetry breaking among the nodes that want to transmit, to prevent them from transmitting at the same time. Simple solutions like *Time Division Multiple Access* (TDMA), which assigns predetermined slots according to node IDs, are expensive in situations where not all of the nodes want to transmit, since their costs depend on the total number of nodes rather than on the number of actual transmitters. One can improve this solution by allowing nodes that cannot interfere with each other to share a slot, so that the number of slots depends on the node degrees in the network graph rather than on the total number of nodes. However, this requires the nodes to have information regarding the topology of the network, and is still expensive in cases where the contention is less than the node degrees. One successful approach for avoiding collisions in wireless networks is to allow each node to use a schedule of probabilities to decide whether to transmit at each time slot [4,10,28]. These algorithms typically guarantee a high probability of successful transmissions after some bounded number of attempts.

In this paper we provide a coding framework for coping with collisions in a wireless communication model abstraction called the *finite-field additive radio network model*, where a collision of transmissions optimally coded for an *Additive White Gaussian Noise (AWGN)* channel with multiple-user interference is represented to be equivalent to the element-wise XOR of a string of bits representing the original transmissions. More generally, collisions can be modelled as being equivalent to the sum, symbol-wise, of the elements of vectors over a finite field, where the transmission of a user is represented as a vector in that finite field, the XOR case being the special case where the field is $\mathbb{F}_2$. Such a model has been shown to be approximately valid in a high SNR (signal-to-noise ratio) regime, abstracting away the effect of noise in the channels and allowing us to concentrate on the theoretical aspects of the interference among transmissions. Such a model in effect replaces the traditional information-theoretic setting of Gaussian inputs for an AWGN channel with an approximate finite algebraic construct [3,24]. Such additive models have been shown, albeit without a finite field construct, to be effective in high SNR settings even in the absence of underlying capacity-achieving codes, for instance in such systems as zig-zag decoding [16], which can be modelled algebraically [36].

In this additive model, our key observation is that *only if not all messages are valid transmissions then a sum representing a collision might indeed be uniquely decodable.* However, we do not wish to restrict the information the users may send in the wireless system. Instead, we propose encoding the information sent into restricted sets of signals that *do* allow unique decoding when they collide. A node receiving a collision can then uniquely decode the signal to obtain the original unrestricted messages. Clearly, for information-theoretic reasons, we cannot hope to restrict the transmissions without the cost of some overhead in the amount of information sent. The challenge, then, is to find codes that allow unique decoding in the above setting with the shortest possible codewords. Under our high SNR assumption, we consider both half-duplex (sometimes termed time-division duplex - TDD) and full-duplex channels. While the TDD model is by far the most common current mode of operation, high SNR conditions can allow full-duplex operation.

1.1 Our Contributions

The Bounded-Contention Coding (BCC) Framework: We define a new class of codes, which are designed for settings in which the number of transmitters is bounded by a known constant a. Each such code consists of an individual code for each node, and has the desirable property that, when codewords from at most a different transmitting nodes are summed up, the result can be uniquely decoded into the original transmissions. This decoding process does not require that the nodes know the identities of the transmitters. Moreover, the active nodes may change from round to round. We show simple constructions of Bounded-Contention Codes, where the length of the codewords depends on both the known contention bound and the total number of nodes, but the dependency on the total number of nodes is only logarithmic.

Distributed computation Using BCC: Using the new Bounded-Contention Coding technique, we show how to obtain local and global broadcast in both single-hop and multi-hop networks. BCC enables *deterministic* local broadcast in a network with n nodes and at most a transmitters with information of ℓ bits each within $O(a \log n + a\ell)$ bits of communication with full-duplex radios, and $O((a \log n + a\ell)(\log n))$ bits, with high probability, with half-duplex radios. When combined with random linear network coding, BCC gives *global* broadcast within $O((D + a + \log n)(a \log n + \ell))$ bits. These results also hold in highly dynamic networks that can change arbitrarily over time under the control of a worst-case adversary.

Further, we show how to remove the assumption that the nodes know a bound a on the contention, by developing a method for handling unknown contention (or contention that varies over space and time), which is common in wireless networks. Also, while it may be reasonable to assume a bound on the contention, it is often the case that the *actual* contention is much smaller.

Because of space limitations, many proofs have been omitted and can be found in the full version of this manuscript.

1.2 Related Work

The finite-field additive radio network model of communication considered in this paper, where collisions result in an addition, over a finite field, of the transmitted signals, was previously studied in [3,24], where the main attention was towards the capacity of the network, i.e., the amount of information that can be reliably transmitted in the network. While the proof of the validity of the approximation [3] is subtle, the intuition behind this work can be readily gleaned from a simple observation of the Cover-Wyner multiple access channel capacity region. Under high SNR regimes, the pentagon of the Cover-Wyner region can, in the limit, be decomposed into a rectangle, appended to a right isosceles triangle [24]. The square can be interpreted as the communication region given by the bits that do not interfere. Such bits do not require special attention. In the case where the SNRs at the receiver for the different users are the same, this rectangle vanishes. The triangular region is the same capacity region as for noise-free additive multiple access channel in a finite field [13], leading naturally to an additive model over a finite field.

Note that, while we consider an equivalent additive finite-field additive model, this does not mean our underlying physical network model is reliant on symbol-wise synchronization between the senders. Asynchrony among users does not affect the behavior of the underlying capacity region [20], on which the approximate model is predicated. Nor are users required to have the same received power in order to have the finite-filed equivalence hold – differences in received power simply lead to different shapes of the Cover-Wyner region, but the interpretation of the triangular and rectangular decomposition of the Cover-Wyner region is not affected. Moreover, our assumption of knowing the interfering users is fairly standard in multiple access wireless communications. Issues of synchronization, SNR determination and identification of users are in practice handled often jointly, since a signature serves for initial synchronization in acquiring the signal of a user, for measuring the received SNR and also for identification of the transmitting user. Finally note that, as long as we have appropriate coding, then the Cover-Wyner region represents the region not only for coordinated transmissions, but also for uncoordinated packetized transmissions, such as exemplified in the classical ALOHA scheme [34]. This result, which may seem counterintuitive, is due in effect to the fact that the system will be readily shown to be stable as long as the individual and sum rates of the Cover-Wyner region will exceed the absolute value of the derivative of an appropriately defined Lyapunov function based on the queue length of a packetized ALOHA system.

There has been work on optimization of transmissions over the model of [3]. These approaches [2,15,38] generally provide algorithms for code construction or for finding the maximum achievable rate, for multicast connections, over a high SNR network under the model of [3]. The approach of [14,24,25] considers a more general finite-field model and reduces the problem to an algebraic network coding problem [26]. Random code constructions, inspired from [19] are then with high probability optimal. These approaches differ from our work in this paper in that they are interested in throughput maximization in a static model rather than completion delay when multiple transmission rounds may occur. We are interested in the latter model and, in particular, in how long it takes to broadcast successfully a specific piece of information.

There are many deterministic and randomized algorithms for scheduling transmissions in wireless networks. They differ in some aspects of the model, such as whether the nodes can detect collision or cannot distinguish between a collision and silence, and whether the nodes know the entire network graph, or know only their neighbors, or do not have any such knowledge at all. Some papers that studied local broadcast are [11,27], where deterministic algorithms were presented, and [6,21,33], which studied randomized algorithms.

In the setting of a wireless network, deterministic global broadcast of a single message was studied in [10,12,28], the best results given being $O(n \log n)$ and $O(n \log^2 D)$, where D is the diameter of the network. Bar-Yehuda et al. [4] were the first to study randomized global broadcast algorithms. Kowalski and Pelc [28] and Czumaj and Rytter [10] presented randomized solutions based on selecting sequences, with complexities of $O(D \log \frac{n}{D} + \log^2 n)$. These algorithms match lower bounds of [1,32] but in a model that is weaker than the one addressed in this paper. The algorithms mentioned above are all for global broadcast of one message from a known source. For multiple messages, a deterministic algorithm for k messages with complexity $O(k \log^3 n + n \log^4 n)$ appears in [7], while randomized global broadcast of multiple messages was studied in [5,22,23]. We refer the reader to an excellent survey on broadcasting in radio networks in [37].

Wireless networks are not always static; for example, nodes may fail, as a result of hardware or software malfunctions. Tolerating failed and recovered components is a basic challenge in decentralized systems because the changes to the network graph are not immediately known to nodes that are some distance away. Similarly, nodes may join and leave the network, or may simply be mobile. All of these cases result in changes to the network graph that affect communication. Depending on the assumptions, these changes can be quite arbitrary. Having a dynamic network graph imposes additional challenges on designing distributed algorithms for wireless networks. Dynamic networks have been studied in many papers. The problems addressed include determining the number of nodes in the network, gossiping messages, data aggregation, and distributed consensus [9,29–31]. For global broadcast, some papers assume restrictions on the changes made in each round. For example, [8] consider graph changes that are random. They also consider the worst-case adversary, as do the studies in [29,35]. In [29] collisions are abstracted away, so that edges of the network graph do not represent nodes that hear the transmissions, but nodes that actually obtain the message. In [17], the authors show how to use network coding in order to obtain more efficient algorithms for global broadcast in this dynamic model.

2 Network Abstraction

We consider a wireless network where the transmission of a node is received at all neighboring nodes, perhaps colliding with transmissions of other nodes. Formally, the network is represented by an undirected graph $G = (V, E)$, where $|V| = n$. We denote by $N(u)$ the subset of V consisting of all of u's neighbors in G and by D the diameter of the network. The network topology is unknown.

We address two different radio models. One is the full-duplex model, in which nodes can listen to the channel while transmitting. The second is the half-duplex mode, in which at every time, a node can either transmit or listen to the channel. A transmission of a node $v \in V$ is modeled as a string of bits $\bar{s}_v$. The communication abstraction is such that the information received by a listening node $u \in V$ is equal to $\bigoplus_{v \in N(u)} \bar{s}_v$, where the operation $\oplus$ is the bit-wise XOR operation.

The model is further assumed to be synchronous, that is, the nodes share a global clock, and a fixed slot length (typically $O(\text{polylog}\, n)$) is allocated for transmission.

Most of the paper assumes a bound $a \leq n$ on the contention that is known to all nodes. However, the actual contention in the network, which we denote by a', may be even smaller than a. Each node has a unique ID from some set I of size $|I| = N$, such that $N = n^{O(1)}$.

3 Bounded-Contention Codes

To extract information from collisions, we propose the following coding technique for basic Bounded-Contention Coding, in which each node encodes its message into a codeword that it transmits, in such a way that a collision admits only a single possibility for the set of messages that produced it. This enables unique decoding.

Definition 1. *An $[M, m, a]$-BCC-code is a set $C \subseteq \{0,1\}^m$ of size $|C| = M$ such that for any two subsets $S_1, S_2 \subseteq C$ (with $S_1 \neq S_2$) of sizes $|S_1|, |S_2| \leq a$ it holds that $\bigoplus S_1 \neq \bigoplus S_2$, where $\bigoplus X$ of $X = \{\bar{x}_1, \ldots, \bar{x}_t\}$ is the bit-wise XOR $\bar{x}_1 \oplus \cdots \oplus \bar{x}_t$.*

As a warm-up, we start by giving an example of a very simple BCC-code. This is the code of all unit vectors in $\{0,1\}^M$, i.e., $C = \{\bar{x}_i | 1 \leq i \leq M, \bar{x}_i(j) = 1$ if and only if $i = j\}$. It is easy to see that C is an $[M, M, M]$-BCC-code, since every subset $S \subseteq C$ is of size $s \leq M$, and we have $\bigoplus S = \bar{x}_s$, where $\bar{x}_s(j) = 1$ if and only if $\bar{x}_i \in S$, implying that Definition 1 holds.

The parameter M will correspond to the number of distinct transmissions possible throughout the network; for example, it could correspond to the number of nodes. The parameters a and m will correspond to contention and slot length, respectively. Therefore, the BCC-codes that interest us are those with m as small as possible, in order to minimize the amount of communication. The above simple code, although tolerating the largest value of a possible, has a very large codeword length. Hence, we are interested in finding BCC codes that trade off between a and m. To show that such good codes exist, we need the following basic background on linear codes. An $[M, k, d]$-*linear code* is a linear subspace of $\{0,1\}^M$, (any linear combination of codewords is also a codeword) of dimension k and minimum Hamming weight d (the Hamming weight of a codeword is the number of indexes in which the codeword differs from zero). The *dual code* of a linear code D, denoted $D^\perp$, is the set of codewords that are orthogonal to all codewords of D, and is also a linear code. It holds that $(D^\perp)^\perp = D$. The construction for arbitrary $[M, m, a]$-BCC-codes works as follows.

BCC Construction: Let D be a linear code of words with length M and Hamming weight at least $2a + 1$. Let $\{\bar{x}_1, \ldots, \bar{x}_m\}$ be a basis for the dual code $D^\perp$ of D. Let H be the $m \times M$ parity-check matrix of D, i.e., the matrix whose rows are $\bar{x}_1, \ldots, \bar{x}_m$, and let C be the set of columns of H. We claim that C is the desired BCC-code.

Lemma 1. *The code C constructed above is an $[M, m, a]$-BCC-code.*

As the following sections will show, we need $[M, m, a]$-BCC-codes with m as small as possible and a as large as possible. By Lemma 1, this means we need to find linear codes of dimension $k = M - m$ as large as possible and minimum Hamming weight $d \geq 2a + 1$ as large as possible. Note that we are only interested in the existence of good codes as the ID of a node will imply the codewords assigned to it, requiring no additional communication.

Lemma 2. *There is an $[M, m, a]$-BCC code with $m = O(a \log M)$.*

Lemma 2 implies, for example, that there are BCC-codes with $a = \Theta(\log M)$ and $m = O(\log M \cdot \log M) = O(\log^2 M)$. As explained earlier, for solving the problem of local broadcast, the parameters a and m correspond to the contention and the transmission length, respectively. As we show in the next section, such BCC-codes with polylogarithmic parameters are well-suited for the case of bounded contention, hence we refer to them as Bounded-Contention Codes.

In fact, the BCC-codes presented above are optimal, since $\Omega(a \log M)$ is a lower bound for m. The reason for this is that each XOR needs to uniquely correspond to a subset of size at most a out of a set of size M. The number of such subsets is $\binom{M}{a}$, therefore each codeword needs to have length $\Omega(\log \binom{M}{a}) = \Omega(a \log M)$.

4 Local Broadcast

This section shows how to use BCC-codes for obtaining local broadcast in the additive radio network model. The simplest way to illustrate our technique is the following. Assume that in every neighborhood there are at most a participants, and each node needs to learn the IDs of all participants in its neighborhood. The nodes use an $[N, a \log N, a]$-BCC code to encode their IDs and, since at most a nodes transmit in every neighborhood, every receiver is guaranteed to be able to decode the received XOR into the set of local participants. For the case of a single-hop network, we show in the full version of this paper how this information can then be used in order to assign unique transmission slots for the participants. However, while this shows the simplicity of using BCC-codes for coping with collisions, it does not extend to multi-hop networks since these require more coordination in order to assign slots for interfering transmitters (who can be more than a single hop from one another, as in the case of a *hidden terminal*). Instead, we show how to use BCC-codes for directly coding the information rather than only the IDs of the transmitters.

We assume the real data that a node v wants to transmit may be any element $s \in \{0, 1\}^\ell$. Instead of encoding the IDs of the nodes in the network, we use an $[N2^\ell, m, a]$-BCC code C and every node v is assigned 2^ℓ codewords $\{C(v, s) | s \in \{0, 1\}^\ell\}$ for it to use. This implies that the length of the codewords is $m = O(a \log (N2^\ell)) = O(a(\log N + \ell)) = O(a(\log n + \ell))$. Notice that this is optimal since $a \log n$ is required in order to distinguish subsets of size a among n nodes, and the $a\ell$ term cannot be avoided if a nodes transmit ℓ bits each.

With half-duplex radios, we let each node choose whether it listens or transmits (if needed) with probability 1/2. This gives that for every message and every node v, in each round there is probability 1/4 for the message to be transmitted and heard by v. In expectation, a constant number of rounds is needed for v to hear any single message, and using a standard Chernoff bound implies that $O(\log n)$ rounds are needed with high probability. Finally, a union bound over all n nodes and all messages gives the following theorem.

Theorem 1. *In a multi-hop network with at most a transmitters with information of ℓ bits each in each $N(u)$, local broadcast can be obtained within $O((a \log n + a\ell) \log n)$ bits, with high probability.*

5 Global Broadcast

In this section we show how to obtain global broadcast by combining BCC and network coding. We assume that at most a nodes have a message of ℓ bits each that needs to be received by all nodes of the network. We first briefly introduce random linear network coding (RLNC) as a solution to the global broadcast problem in additive radio networks and then, in Subsection 5.2, show how BCC can significantly reduce the coding coefficient overhead of RLNC when $a << n$.

5.1 Random Linear Network Coding

RLNC is a powerful method to achieve optimal global broadcast, in particular in distributed networks in which nodes cannot easily coordinate to route information through the network. Instead of sending pieces of information around directly, RLNC communicates (random) linear combinations of messages over a finite field F_q. In this paper we will choose the field size q to be 2 which allows us to see vectors in F_q simply as bit-vectors and linear combinations of vectors as XORs.

We denote with $m_u \in F_2^\ell$ the message sent out by node u and denote with S the set of at most a nodes that initially have a message. Given this, any packet sent out during the RLNC protocol has the form $(\mu, \sum_{u \in I} \mu_u m_u) \in F_2^{N+\ell}$ where $\mu \in F_2^N$ is a coefficient vector indicating which messages are XOR-ed together in the second portion of a packet, i.e., a characterizing vector. We call packets of this form *valid*. A node u that initially starts with a message m_u treats this message as if it received the packet (e_u, m_u) before round one, where e_u is the standard basis vector corresponding to u (that is, with a one at the coefficient corresponding to u, and zeros otherwise). During the protocol, each node that is supposed to send a packet takes all packets it has received so far and includes each of them independently with probability 1/2 in the new packet. The new packet is formed by taking the XORs of all packets selected in this way (if no packet is selected the node remains silent or alternatively sends the all zero vector). Nodes decode by using Gaussian elimination. This can be done if and only if a node has received a valid packets with linearly independent coefficient vectors.

We note that, because of linearity, all initial packets and all packets created during the RLNC protocol are valid. More importantly, if multiple neighbors of a node send valid packets then the XOR of these packets which is received is also valid since the coefficient vectors and the message part XOR separately and component-wise. This makes RLNC a simple but powerful tool for exploiting the linear and additive nature of the additive radio networks we study in this paper.

We analyze the complexity of this RLNC scheme when used on top of an additive radio network. As in Section 4, nodes can either transmit or listen to the channel at any given round since they have half-duplex radios. We use the above RLNC algorithm together with the strategy of choosing in each round whether to transmit or listen at random with probability $1/2$.

We show that the RLNC protocol achieves an optimal round complexity of $O(D + a + \log n)$ with high probability. Our proof is based on the projection analysis technique from [18] but we give a simple, self-contained proof in the full version of this paper. The reason that the analysis carries over from a message passing model to the radio networks considered here so easily is their additivity. In particular, we use the effect that the XOR of randomly selected packets sent out by several neighbors which get XORed in the air are equivalent to the XOR of a random selection of packets known to at least one neighbor.

Theorem 2. *RLNC disseminates all a messages, with high probability, in $O(D+a+\log n)$ rounds in which messages of $N+\ell$ bits are sent in each round.*

5.2 Reducing the Overhead of Random Linear Network Coding via BCC-Codes

Note that Theorem 2 shows that RLNC has an essentially optimal round complexity. In particular, $\Omega(D)$ is a trivial lower bound since information passes at most one step in the network per round and $\Omega(a)$ is a lower bound too since in each round at most ℓ bits worth of information messages are received while $a\ell$ bits need to be learned in total. Lastly, the $\Omega(\log n)$ factor is tight for the proposed algorithm, too, because of the randomness used. On the other hand, the packets sent around have size $N+\ell$ while carrying only ℓ bits of information about the messages. Note that $N > n$ and in many cases $N = n^c >> \ell$ for some constant c which renders the standard RLNC implementation highly inefficient.

The reason for this is that we use an N bit vector as a header to describe the set of IDs of nodes whose message is coded into the packet. This vector is always extremely sparse since at most $n << N$ nodes are present and at most $a << n$ nodes are sending a message. Instead of writing down a vector as is one could thus try to use a short representation of these sparse vectors. Writing down only the IDs of the non-zero components would be such a sparse representation (with almost optimal bit size $a \log N$) but does not work here, because when multiple neighbors of a node send sparse coefficient vectors their received XOR cannot be uniquely decoded. BCC-codes solve exactly this problem, by providing a *sparse vector representation*:

Definition 2. *Let I be an ID set of size N and a be a sparseness parameter. Any $[N, a\log N, a]$-BCC code C mapping any ID $u \in I$ to $C(u) \in F_2^{a\log N}$ induces a* sparse vector representation *s that maps the vector $\mu \in F_2^N$ to $s(\mu) = \sum_{u|\mu_u=1} C(u)$.*

The following two properties make this representation so useful (in particular in this context):

Lemma 3. *Let I be an ID set of size N, a be a sparseness parameter and s be a sparse vector representation induced by a BCC-code C. For any two vectors $\mu, \mu' \in F_2^N$ with at most a non-zero components we have:*

- *Unique Decodability: $\mu \neq \mu' \implies s(\mu) \neq s(\mu')$.*
- *Homomorphism under addition: $s(\mu) + s(\mu') = s(\mu + \mu')$.*

Replacing the coefficient vectors μ in the RLNC scheme with their sparse representation leads to the much more efficient RLNC+BCC scheme.

As in the RLNC protocol, we denote with $m_u \in F_2^\ell$ the message sent out by node u and denote with S the set of at most a nodes that initially have a message. Any packet sent out during the RLNC+BCC protocol has the form $(\mu, \sum_{u\in I} \mu_u m_u) \in F_2^{a\log N+\ell}$ where $\mu \in F_2^{a\log N}$ is a *coded* coefficient vector indicating which messages are XOR-ed together in the second portion of a packet, i.e., it holds the XOR of the BCC-codewords of the IDs of the messages. As before, each node that is supposed to send a packet takes all packets it has received so far and includes each of them independently with probability $1/2$ in the new packet. The new packet is formed by taking the XORs of all packets selected in this way, preceded by the corresponding coded coefficient vector. Nodes decode by using Gaussian elimination, which can be done if and only if a node has received a valid packets with linearly independent coefficient vectors. We note that, because of linearity, all initial packets and all packets created during the RLNC protocol are valid. Unlike having a list of IDs as a sparse representation of the coefficient vector, the power of BCC here is that if multiple neighbors of a node send valid packets then the XOR of these packets which is received is also valid since the BCC-coded coefficient vectors and the message part XOR separately and component-wise. Formally, the algorithm is identical to RLNC, except that the set S_u of messages received by node u is initialized to $(C(u), m_u)$ instead of (e_u, m_u), and the node listens for $a\log N + \ell$ bits, rather than $N + \ell$.

Theorem 3. *RLNC+BCC disseminates all a messages, with high probability, in $O(D + a + \log n)$ rounds in which messages of $O(a\log n + \ell)$ bits are sent in each round.*

6 Dynamic Networks

In this section, we consider the case of a highly-dynamic network with a worst-case adversary: in every round, that is, between the times nodes send packets, the network graph is determined by the adversary, which observes the entire computation so far when deciding upon the graph for the next round. Notice that all of the above results for local broadcast hold for such dynamic networks, given that a slot length is sufficiently long in order to contain the required information. This is, for example, $O(a(\log n + \ell))$ bits in the full-duplex model, which is reasonable to assume if a and ℓ are not too large. We get absolute guarantees for local broadcast in this highly dynamic setting, while existing work on avoiding collisions in radio networks cannot achieve this since they require probabilistic transmissions.

Next, we generalize the RLNC+BBC framework for the case of this highly-dynamic network with a worst-case adversary. The only restriction is that the graph has to be connected in every round. The proof of the resulting theorem is essentially the same as for the static case but instead of arguing that every message makes progress over a shortest path P, we argue that it makes some progress since the graph is always connected. Hence D is replaced by n in the number of rounds needed.

Theorem 4. *In a dynamic additive radio network controlled by an adaptive adversary subject to the constraint that the network is connected at every round, RLNC+BCC achieves global broadcast of a messages, with high probability, in $O(n + a + \log n)$ rounds using a packet size of $O(a \log n + \ell)$ bits.*

7 Estimating the Contention

We have given an almost optimal scheme for achieving global broadcast when the number of senders a (or a good upper bound on it) is known a priori. This assumption is not an unreasonable one, for example, if network statistics show such behavior. However, in many cases, local contention may differ at different places throughout the network, or vary over time. It may also be that the known bound is pessimistic, and the actual contention is much smaller than this bound. In this section, we show a method for removing this assumption by using BCC-codes to quickly determine a (and also reveal the identity of all senders).

The mechanism we present allows for estimating the current contention and then using a code that corresponds to that estimate. A standard way to obtain a good estimation of contention is by having the nodes double a small initial guess until they succeed in local broadcast. In our BCC framework, the tricky part of this approach is to identify success. Specifically, using a bounded-contention code with parameter a for a set S of $k > a$ transmitters may produce an XOR that is a valid XOR of some set S' of $k' \leq a$ transmitters. Hence, the nodes need to be able to distinguish between such a case and the case where S' is the true set of transmitters.

The idea behind our algorithm is simple. We use an $[N, 2k \log N, 2k]$-BCC code to send out IDs in every round to make them propagate through the network. Every node u keeps track of the set S_u of all IDs it has heard from so far. In every round node u sends out an XOR in which each of the IDs in S_u is independently included with probability $1/2$. If $k \geq a$ then nodes receive the sum of at most a nodes in every round and are able to split this sum into IDs which are then added to their sets. This way an ID propagates from one node to the next with constant probability and we show that within $O(D+\log n)$ rounds every node, with high probability, receives the ID of every node that wants to send. However, if $k < a$ we may get XORs of more than $2k$ IDs, which have no unique decoding guarantees by the BCC-code. The following algorithm takes care of this by detecting such a case eventually (and sufficiently fast).

Algorithm 1. Estimating the Contention a, pseudocode for node u.

```
1:  $k \leftarrow 2$
2:  REPEAT UNTIL $fail_u = false$ and $|S_u| \leq k$
3:      $k \leftarrow 2k$
4:      $fail_u \leftarrow false$
5:      $C \leftarrow [N, 2k \log N, 2k]$-BCC code
6:      IF node $u$ is a sender
7:          $S_u \leftarrow \{C(u)\}$
8:      ELSE
9:          $S_u \leftarrow \emptyset$
10:     FOR iteration $i = 1, \ldots, 32(D + \log n)$:
11:         IF $fail_u$
12:             send $\log n$ random bits
13:         ELSE
14:             listen for $\log n$ random bits
15:             IF received a non-zero string
16:                 $fail_u \leftarrow true$
17:         With probability 1/2 DO
18:             Send $\sum_{v \in S_u} X_v C(v)$ where $X_v$ are i.i.d. uniformly Bernoulli
19:         OTHERWISE
20:             listen for $2k \log N$ bits
21:             IF what received can be decoded as $\sum_{v \in S} C(v)$ for a $|S| \leq k$
22:                 $S_u \leftarrow S_u \cup S$
23:             ELSE
24:                 $fail_u \leftarrow true$
```

Theorem 5. *With high probability, Algorithm 1 correctly identifies the subset of senders S at every node after a total amount of communication of $O((D + \log n)(a \log n))$ bits.*

One can use this procedure not just to estimate a but also to exploit the fact that it gives the IDs of all senders in order to simplify the RLNC algorithm. For this, we order the IDs of the senders and assign to the i highest node the ith standard basis vector out of the space F_2^a. We then use this as a sparse and concise coefficient vector in the RLNC protocol. This gives the following:

Corollary 1. *After running the BCC-Estimation algorithm, RLNC achieves global broadcast, with high probability, in $O(D+a+\log n)$ rounds in which packets of size $a+\ell$ bits are sent in each round.*

8 Discussion

This paper presents a coding technique for additive wireless networks, which allows efficient local and global broadcast given a bound on the amount of contention. It also shows how to estimate the contention when it is not known in advance. The results hold also for dynamic networks whose arbitrary changes are controlled by a worst-case adversary. For full-duplex radios, it gives a deterministic framework providing absolute guarantees.

Directions for further research include using BCC-codes for solving additional distributed problems in the additive wireless network model, and handling extensions to the model, such as noise and asynchrony.

Acknowledgments. The authors thank Seth Gilbert for useful discussions regarding probabilistic retransmissions, MinJi Kim and Ali ParandehGheibi for many discussions about the XOR collisions model addressed in this paper, and Amir Shpilka for pointing out the existence of the simple codes we use to implement our BCC framework. This work was supported in part by the Simons Postdoctoral Fellows Program and NSF Award 0939370-CCF.

References

1. Alon, N., Bar-Noy, A., Linial, N., Peleg, D.: A lower bound for radio broadcast. Journal of Computer and System Sciences 43, 290–298 (1991)
2. Amaudruz, A., Fragouli, C.: Combinatorial algorithms for wireless information flow. In: SODA (2009)
3. Avestimehr, A.S., Diggavi, S.N., Tse, D.N.C.: Wireless network information flow: A deterministic approach. IEEE Transactions on Information Theory 57(4), 1872–1905 (2011)
4. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. J. Comput. Syst. Sci. 45(1), 104–126 (1992)
5. Bar-Yehuda, R., Israeli, A., Itai, A.: Multiple communication in multi-hop radio networks. SIAM Journal on Computing 22, 875–887 (1993)
6. Bienkowski, M., Klonowski, M., Korzeniowski, M., Kowalski, D.R.: Dynamic sharing of a multiple access channel. In: Marion, J.-Y., Schwentick, T. (eds.) 27th International Symposium on Theoretical Aspects of Computer Science (STACS 2010). Leibniz International Proceedings in Informatics (LIPIcs), vol. 5, pp. 83–94. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl (2010)
7. Chlebus, B.S., Kowalski, D.R., Pelc, A., Rokicki, M.A.: Efficient Distributed Communication in Ad-Hoc Radio Networks. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 613–624. Springer, Heidelberg (2011)
8. Clementi, A.E.F., Monti, A., Pasquale, F., Silvestri, R.: Broadcasting in dynamic radio networks. J. Comput. Syst. Sci. 75(4), 213–230 (2009)
9. Cornejo, A., Newport, C.: Prioritized gossip in vehicular networks. In: Proceedings of the 6th ACM SIGACT/SIGMOBILE International Workshop on Foundations of Mobile Computing (DIALM-POMC 2010), Cambridge, MA (September 2010)
10. Czumaj, A., Rytter, W.: Broadcasting algorithms in radio networks with unknown topology. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2003), p. 492 (2003)

11. Czyzowicz, J., Gasieniec, L., Kowalski, D.R., Pelc, A.: Consensus and mutual exclusion in a multiple access channel. IEEE Transactions on Parallel and Distributed Systems 22(7), 1092–1104 (2011)
12. De Marco, G.: Distributed broadcast in unknown radio networks. SIAM Journal of Computing 39, 2162–2175 (2010)
13. Effros, M., Médard, M., Ho, T., Ray, S., Karger, D., Koetter, R.: Linear network codes: A unified framework for source channel, and network coding. In: Proceedings of the DIMACS workshop on Network Information Theory (2003) (invited paper)
14. Erez, E., Xu, Y., Yeh, E.M.: Coding for the deterministic network model. In: Allerton Conference on Communication, Control and Computing (2010)
15. Goemans, M.X., Iwata, S., Zenklusen, R.: An algorithmic framework for wireless information flow. In: Proceedings of Allerton Conference on Communication, Control, and Computing (2009)
16. Gollakota, S., Katabi, D.: Zigzag decoding: combating hidden terminals in wireless networks. In: SIGCOMM, pp. 159–170 (2008)
17. Haeupler, B., Karger, D.: Faster information dissemination in dynamic networks via network coding. In: Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing (PODC 2011), San Jose, CA, pp. 381–390 (June 2011)
18. Haeupler, B.: Analyzing network coding gossip made easy. In: Proceedings of the 43rd Annual ACM Symposium on Theory of Computing, STOC 2011, pp. 293–302. ACM, New York (2011)
19. Ho, T., Médard, M., Koetter, R., Karger, D.R., Effros, M., Shi, J., Leong, B.: A random linear network coding approach to multicast. IEEE Transactions on Information Theory 52(10), 4413–4430 (2006)
20. Hui, J., Humblet, P.: The capacity region of the totally asynchronous multiple-access channel. IEEE Transactions on Information Theory 31(2), 207–216 (1985)
21. Jurdziński, T., Stachowiak, G.: Probabilistic Algorithms for the Wakeup Problem in Single-Hop Radio Networks. In: Bose, P., Morin, P. (eds.) ISAAC 2002. LNCS, vol. 2518, pp. 535–549. Springer, Heidelberg (2002)
22. Khabbazian, M., Kowalski, D.: Time-efficient randomized multiple-message broadcast in radio networks. In: Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing (PODC 2011), San Jose, California, June 6-8 (2011)
23. Khabbazian, M., Kowalski, D., Kuhn, F., Lynch, N.: Decomposing broadcast algorithms using Abstract MAC layers. In: Proceedings of Sixth ACM SIGACT/SIGMOBILE International Workshop on Foundations of Mobile Computing (DIALM-POMC 2010), Cambridge, MA (September 2010)
24. Kim, M., Erez, E., Edmund, M., Médard, M.: Deterministic network model revisited: An algebraic network coding approach. CoRR, abs/1103.0999 (2011)
25. Kim, M., Médard, M.: Algebraic network coding approach to deterministic wireless relay network. In: Allerton Conference on Communication, Control and Computing (2010)
26. Koetter, R., Médard, M.: An algebraic approach to network coding. IEEE/ACM Transaction on Networking 11, 782–795 (2003)
27. Komlós, J., Greenberg, A.G.: An asymptotically fast nonadaptive algorithm for conflict resolution in multiple-access channels. IEEE Transactions on Information Theory 31(2), 302–306 (1985)
28. Kowalski, D.R., Pelc, A.: Kowalski and Andrzej Pelc. Broadcasting in undirected ad hoc radio networks. Distribed Computing 18, 43–57 (2005)

29. Kuhn, F., Lynch, N.A., Oshman, R.: Distributed computation in dynamic networks. In: Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC), pp. 513–522 (2010)
30. Kuhn, F., Moses, Y., Oshman, R.: Coordinated consensus in dynamic networks. In: Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, San Jose, California, June 6-8 (2011)
31. Kuhn, F., Oshman, R.: The Complexity of Data Aggregation in Directed Networks. In: Peleg, D. (ed.) Distributed Computing. LNCS, vol. 6950, pp. 416–431. Springer, Heidelberg (2011)
32. Kushilevitz, E., Mansour, Y.: An $\Omega(D \log(N/D))$ lower bound for broadcast in radio networks. In: Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC 1993), pp. 65–74. ACM, New York (1993)
33. Martel, C.U.: Maximum finding on a multiple access broadcast network. Information Processing Letters 52(1), 7–15 (1994)
34. Medard, M., Huang, J., Goldsmith, A.J., Meyn, S.P., Coleman, T.P.: Capacity of time-slotted aloha packetized multiple-access systems over the awgn channel. IEEE Transactions on Wireless Communications 3(2), 486–499 (2004)
35. O'Dell, R., Wattenhofer, R.: Information dissemination in highly dynamic graphs. In: Proceedings of the 2005 Joint Workshop on Foundations of Mobile Computing, DIALM-POMC 2005, pp. 104–110. ACM, New York (2005)
36. ParandehGheibi, A., Sundararajan, J.-K., Médard, M.: Collision helps - algebraic collision recovery for wireless erasure networks. In: WiNC (2010)
37. Peleg, D.: Time-Efficient Broadcasting in Radio Networks: A Review. In: Janowski, T., Mohanty, H. (eds.) ICDCIT 2007. LNCS, vol. 4882, pp. 1–18. Springer, Heidelberg (2007)
38. Shi, C., Ramamoorthy, A.: Improved combinatorial algorithms for wireless information flow. In: Proceedings of Allerton Conference on Communication, Control, and Computing (2010)

Distributed Backbone Structure for Algorithms in the SINR Model of Wireless Networks*

Tomasz Jurdzinski[1] and Dariusz R. Kowalski[2]

[1] Institute of Computer Science, University of Wrocław, Poland
[2] Department of Computer Science, University of Liverpool, United Kingdom

Abstract. The Signal-to-Interference-and-Noise-Ratio (SINR) physical model is one of the most popular models of wireless networks. Despite of the vast amount of study done in design and analysis of centralized algorithms supporting wireless communication under the SINR physical model, little is known about distributed algorithms in this model, especially deterministic ones. In this work we construct, in a deterministic distributed way, a backbone structure on the top of a given wireless network, which can be used for efficient transformation of many algorithms designed in a simpler model of ad hoc broadcast networks without interference into the SINR physical model with uniform power of stations. The time cost of the backbone data structure construction is only $O(\Delta \operatorname{polylog} N)$ rounds, where Δ is roughly the network density and $\{1, \ldots, N\}$ is the range of identifiers (IDs) and thus N is an upper bound on the number of nodes in the whole network. The core of the construction is a novel combinatorial structure called SINR-selector, which is introduced in this paper. We demonstrate the power of the backbone data structure by using it for obtaining efficient $O(D + \Delta \operatorname{polylog} N)$ round and $O(D + k + \Delta \operatorname{polylog} N)$ round deterministic distributed solutions for leader election and multi-broadcast, respectively, where D is the network diameter and k is the number of messages to be disseminated.

Keywords: Wireless networks, SINR, Backbone structure, Distributed algorithms, Leader Election, Multi-message broadcast.

1 Introduction

In this work we study a fundamental problem how to transform algorithms designed and analyzed for ad hoc networks without any interference (e.g., message passing or multicast networks) to ad-hoc wireless networks under the Signal-to-Interference-and-Noise-Ratio model (SINR). A wireless network considered in this work consists of n stations, also called nodes, with uniform transmission powers, deployed in the two-dimensional Euclidean space. Stations act in synchronous rounds; in every communication round a station can either transmit

* This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/G023018/1]. The work of the first author was done during this author stay at the University of Liverpool. The work of the second author was done during this author visit at the Institute IMDEA Networks, Madrid, Spain.

a message or listen to the wireless medium. A communication (or reachability) graph of the network is the graph defined on network nodes and containing links (v, w) such that if v is the only transmitter in the network then w receives the message transmitted by v under the SINR physical model. Each station initially knows only its own unique ID in the range $\{1, \ldots, N\}$, location,[1] and parameters N and Δ, where Δ is the upper bound on the node degree in the communication graph of the network, and corresponds roughly to the network density.

We consider global communication tasks in the SINR wireless setting, and our objective is to minimize time complexity (i.e., the number of rounds) of deterministic distributed solutions. In order to overcome the impact of signal interference, we show how to compute a backbone data structure in a distributed deterministic way, which also clusters nodes and implements efficient inter- and intra-cluster communication. It allows to transform a variety of algorithms designed and analyzed in models without interference to the SINR wireless model with only additive $O(\Delta \operatorname{polylog} N)$ overhead on time complexity. We demonstrate such efficient transformation on two tasks: leader election and multi-broadcast with small messages.

1.1 Previous and Related Results

SINR Model. The Signal-to-Interference-and-Noise-Ratio (SINR) physical model is currently the most popular framework for modelling physical wireless interference for the purpose of design and theoretical analysis of wireless communication tasks. There is a vast amount of work on centralized algorithms under the SINR model. The most studied problems include connectivity, capacity maximization, link scheduling types of problems (e.g., [10,16,1]). See also the survey [13] for recent advances and references.

Recently, there is a growing interest in developing solutions to *local communication* problems, in which stations are only required to exchange information with a subset of their neighbors. Examples of such problems include local broadcast or local leader election. A deterministic *local* broadcasting, in which nodes have to inform only their neighbors in the corresponding reachability graph, was studied in [25]. The considered setting allowed power control by algorithms, in which, in order to avoid collisions, stations could transmit with any power smaller than the maximal one. Randomized solutions for contention resolution [17] and local broadcasting [12] were also obtained. Single hop topologies were also studied recently under the SINR model, c.f., [22].

Randomized distributed solutions to local communication problems, as cited above, can often be used as a basic tool for obtaining *randomized solutions* to *global communication* tasks, i.e., tasks requiring information exchange throughout the whole network. Examples of such task include multi-broadcast or leader election. A randomized distributed algorithm for the multi-broadcast problem was recently presented by Yu et al. [24]; this algorithm works under additional assumption that the closest neighbor of a station is in distance at most 1/3 of the

[1] For the result of this work to hold it is enough to know only approximate location.

maximal range of the station. These problems are also related, though often not equivalent, to several graph-related problems of finding a maximal/maximum independent set, minimal/minimum (connected) dominating set (our backbone structure is an example of the latter, with additional useful properties). Recently, an efficient randomized distributed solution was obtained to the problem of finding a constant-density dominating set by Scheideler et al. [23], however the model of that paper is slightly different than the one in this work, i.e., it combines SINR with radio model.

Surprisingly, there are very few results on *deterministic distributed* solutions to fundamental *global communication* problem under the SINR model. To the best of our knowledge, the only result of this type is devoted to data aggregation in networks with carrier sensing ability (which is not given in our model) [14]. Such algorithms are especially important from perspective of maintaining network infrastructure and applications to distributed systems. Deterministic solutions are often desired in such cases, due to their reliability. One could apply a naive round-robin algorithm to build a collision-free deterministic solution to such problems, but apparently it is extremely inefficient.

Radio Network Model. In the related *radio model* of wireless networks, a message is successfully heard if there are no other simultaneous transmissions from the *neighbors* of the receiver in the communication graph. This model does not take into account the real strength of the received signals, and also the signals from the outside of some close proximity. In the geometric ad hoc setting, Dessmark and Pelc [8] were the first who studied global communication problems, mainly broadcasting. They analyzed the impact of local knowledge, defined as a range within which stations can discover the nearby stations. The most related results to this paper are devoted to unit-disk graph radio networks (UDG). Emek et al. [9] designed a broadcast algorithm working in time $O(\min\{D+g^2, D\log g\})$ in UDG radio networks with eccentricity D and granularity g, where eccentricity was defined as the minimum number of hops to propagate the broadcast message throughout the whole network and granularity was defined as the inverse of the minimum distance between any two stations. Leader election problem for geometric radio networks was studied e.g., by Chung et al. [4] in the case of mobile devices.

Communication problems are well-studied in the setting of *graph radio model*, in which stations are not necessarily deployed in a metric space. Due to limited space and the fact that this research area is not directly related to the core of this work, we refer the reader to the recent literature on deterministic [3,6,7,11,18,19]. and randomized [2,6,18,21] solutions.

1.2 Our Results

The main result of this work is a deterministic and distributed algorithm constructing a complex backbone distributed data structure, consisting of a combinatorial structure with local operations on it, that is aimed to support wireless global communication tasks; for brief description of this structure see Section 2, and for detail implementation and properties we refer the reader to Section 4. The construction is in $O(\Delta \operatorname{polylog} N)$ rounds, where $\{1, \ldots, N\}$ is the range of identifiers (IDs) and thus N is an upper bound on the number of nodes in the whole network. The algorithm constructing the backbone network uses a novel concept of SINR-selectors, which are specific efficient schedules for ad hoc one-hop communication. We define them and show that their existence in Section 3. We do not provide efficient construction of SINR-selectors, but there is a randomized method of selecting a schedule that satisfies properties of a SINR-selector with high probability [15].

Our work can be also viewed as a deterministic distributed implementation of an abstract MAC layer, introduced by Kuhn et al. [20], under the SINR model. It also allows to transform several algorithms designed for networks without interference to the SINR wireless model, with two additive overheads: $O(\Delta \operatorname{polylog} N)$ coming from spanning the backbone data structure, and the number of required parallel outer-backbone communication tasks multiplied by $O(\Delta)$. In many cases the number of such parallel local convergecasts can be lowered to $\operatorname{polylog} N$ or even a constant. As examples, we argue that the problems of leader election and multi-broadcast (with small messages) can be solved by a deterministic distributed algorithm in almost optimal $O(D + \Delta \operatorname{polylog} N)$ and $O(k + D + \Delta \operatorname{polylog} N)$ rounds, respectively. They are shown to be optimal in the SINR model up to a polylogarithmic factor, i.e., they require $\Omega(D + \Delta)$ and $\Omega(k + D + \Delta)$ rounds, respectively.

Due to limited space, some proofs and technical details are deferred to the full version of the paper [15].

2 Model and Notation

Throughout the paper, $\mathbb{N}$ denotes the set of natural numbers and $\mathbb{Z}$ denotes the set of integers. For $i, j \in \mathbb{N}$, we use the notation $[i, j] = \{k \in \mathbb{N} \mid i \leq k \leq j\}$ and $[i] = [1, i]$.

We consider a wireless network consisting of n *stations*, also called *nodes*, deployed into a two dimensional Euclidean space and communicating by a wireless medium. All stations have unique integer IDs in the set $[N]$, where N is an integer model parameter. Stations are denoted by letters u, v, w, which simultaneously denote their IDs. Stations are located on the plane with *Euclidean metric* $\operatorname{dist}(\cdot, \cdot)$. Each station v has its *transmission power* P_v, which is a positive real number. There are three fixed model parameters: path loss $\alpha \geq 2$, threshold $\beta \geq 1$, and ambient noise $\mathcal{N} \geq 1$. The $SINR(v, u, \mathcal{T})$ ratio, for given stations u, v and a set of (transmitting) stations $\mathcal{T}$, is defined as follows:

$$SINR(v, u, \mathcal{T}) = \frac{\frac{P_v}{\operatorname{dist}(v,u)^\alpha}}{\mathcal{N} + \sum_{w \in \mathcal{T} \setminus \{v\}} \frac{P_w}{\operatorname{dist}(w,u)^\alpha}} \tag{1}$$

In the *Signal-to-Interference-and-Noise-Ratio model* (SINR) considered in this work, station u *successfully receives*, or *hears*, a message from station v in a round if $v \in \mathcal{T}$, $u \notin \mathcal{T}$, and the following two properties hold:

- $SINR(v, u, \mathcal{T}) \geq \beta$, where $\mathcal{T}$ is the set of stations transmitting at that time,
- $P_v \text{dist}^{-\alpha}(v, u) \geq (1 + \varepsilon)\beta\mathcal{N}$,

where $\varepsilon > 0$ is a fixed *sensitivity parameter* of the model. The above definition is common in the literature, c.f., [17].[2]

In the paper, we make the following assumptions for the sake of clarity of presentation: $\beta = 1$, $\mathcal{N} = 1$. In general, these assumptions can be dropped without harming the asymptotic performances of the presented algorithms and lower bounds formulas.

Ranges and Uniformity. The *communication range* r_v of a station v is the radius of the circle in which a message transmitted by the station is heard, provided no other station transmits at the same time. A network is *uniform* when ranges (and thus transmission powers) of all stations are equal, or *nonuniform* otherwise. In this paper, only uniform networks are considered. For clarity of presentation we make an assumption that all powers are equal to 1, i.e., $P_v = 1$ for each v. The assumed value 1 of P_v can be replaced by any fixed positive value without changing asymptotic formulas for presented algorithms and lower bounds. Under these assumptions, $r_v = r = (1 + \varepsilon)^{-1/\alpha}$ for each station v. The *range area* of a station with range r located at point (x, y) is defined as a ball of radius r centered at (x, y).

Communication Graph and Graph Notation. The *communication graph* $G(V, E)$, also called the *reachability graph*, of a given network consists of all network nodes and edges (v, u) such that u is in the range area of v. Note that the communication graph is symmetric for uniform networks, which are considered in this paper. By a *neighborhood* $\Gamma(u)$ of a node u we mean the set of all neighbors of u, i.e., the set $\{w \mid (w, u) \in E\}$ in the communication graph $G(V, E)$ of the underlying network. The *graph distance* from v to w is equal to the length of a shortest path from v to w in the communication graph (where the length of a path is equal to the number of its edges). The *eccentricity* of a node is the maximum graph distance from this node to all other nodes (note that the eccentricity is of order of the diameter if the communication graph is symmetric — this is also the case in this work).

We say that a station v transmits *c-successfully* in a round t if v transmits a message in round t and this message is received by each station u in a distance smaller or equal to c from v. We say that a station v transmits *successfully* in round t if it transmits r-successfully, i.e., each of its neighbors in the communication graph can successfully receive its message. Finally, v transmits *successfully* to u in round t if v transmits a message in round t and u successfully receives this message.

Synchronization. It is assumed that algorithms work synchronously in rounds, each station can either act as a sender or as a receiver in a round. All stations

[2] The first condition is a straightforward application of the SINR ratio, comparing strength of one of the received signals with the remainder. The second condition enforces the signal to be sufficiently strong in order to be distinguished from the background noise, and thus to be decoded.

are active in the beginning of computation, i.e., they start computation in the same round.

Collision Detection and Channel Sensing. We consider the model without *collision detection*, that is, if a station u does not receive a message in a round t, it gets no information from the physical wireless layer whether any other station was transmitting in that round and about the value of $SINR(v, u, \mathcal{T})$, for any station u, where $\mathcal{T}$ is the set of transmitting stations in round t.

Communication Problems and Complexity Parameters. We consider two global communication problems: leader election and multi-broadcast. *Leader election* is defined in the following way: initially all stations of a network have the same status non-leader and the goal is for all nodes but one to keep this status and for the remaining single node to get the status leader. Moreover, all nodes must learn the ID of the leader.

The *multi-broadcast* problem is to disseminate k distinct messages initially stored at arbitrary nodes, to the entire network (it is allowed that more than one message is stored in a node).

For the sake of complexity formulas, in the analysis we consider the following parameters: n, N, D, Δ where: n is the number of nodes, $[N]$ is the range of IDs, D is the diameter and Δ is the upper bound on the degree of a station in the communication graph. Note that Δ is proportional, due to geometric constraints, to the largest number of stations in a ball with radius equal to the range of a station, and as such corresponds roughly to the network density.

Backbone Data Structure. As a generic tool for these and possibly other global communication tasks, we design a deterministic distributed algorithm building a backbone data structure, which is a connected dominating set (i.e., a connected subgraph such that each node is either in the subgraph or is a neighbor of a node in the subgraph, also called a CDS) with constant degree, constant approximation of the size of a smallest CDS, and diameter proportional to the diameter of the whole communication graph. Additionally, our algorithm organizes all network nodes in a graph of local clusters, and computes efficient schedules for inter- and intra-cluster communication. More precisely: the computed inter-cluster communication schedule works in constant number of rounds and can be done in parallel without a harm from the interference to the final result, similarly as intra-cluster broadcasting schedules; the intra-cluster convergecast operations, although can be done in parallel, require $\Theta(\Delta)$ rounds.

Messages. In general, we assume that a single message sent in the execution of any algorithm can carry a single message (in case of multi-broadcast) and at most logarithmic, in the size of the ID range N, number of control bits.

Knowledge of Stations. Each station knows its own ID, location, and parameters N, Δ. Moreover, a station knows its location on the plane. (Actually, our algorithms work under weaker assumption that a station knows its location with respect to the points of a square grid with the size of a square proportional to the range of stations.) We assume that stations do not know any other information about the topology of the network at the beginning of an execution of an algorithm.

3 Technical Preliminaries and SINR-Selectors

Given a parameter $c > 0$, we define a partition of the 2-dimensional space into square boxes of size $c \times c$ by the grid G_c, in such a way that: all boxes are aligned with the coordinate axes, point $(0,0)$ is a grid point, each box includes its left side without the top endpoint, its bottom side without the right endpoint and does not include its right and top sides. We say that (i,j) are the coordinates of the box with its bottom left corner located at $(c \cdot i, c \cdot j)$, for $i, j \in \mathbb{Z}$. A box with coordinates $(i,j) \in \mathbb{Z}^2$ is denoted $C(i,j)$. As observed in [8,9], the *grid* $G_{r/\sqrt{2}}$ is very useful in design of algorithms for UDG radio networks, provided r is equal to the range of each station. This follows from the fact that $r/\sqrt{2}$ is the largest parameter of a grid such that each station in a box is in the range of every other station in that box. In the following, we fix $\gamma = r/\sqrt{2}$, where $r = (1+\varepsilon)^{-1/\alpha}$, and call G_γ the *pivotal grid*.

Two boxes C, C' are *neighbors* if there are stations $v \in C$ and $v' \in C'$ such that edge (v, v') belongs to the communication graph of the network. For a station v located in position (x,y) on the plane we define its *grid coordinates* with respect to the grid G_c as the pair of integers (i,j) such that the point (x,y) is located in the box $C(i,j)$ of the grid G_c (i.e., $ic \le x < (i+1)c$ and $jc \le y < (j+1)c$). Moreover, $\text{box}(v) = C(i,j)$ for a station v with grid coordinates (i,j). If not stated otherwise, we will refer to grid coordinates with respect to the pivotal grid.

A (classical) *communication schedule* $\mathcal{S}$ of length T wrt $N \in \mathbb{N}$ is a mapping from $[N]$ to binary sequences of length T. A station with identifier $v \in [N]$ *executes* the schedule $\mathcal{S}$ of length T in a fixed period of time consisting of T rounds, when v transmits a message in round t of that period iff the tth position of $\mathcal{S}(v)$ is equal to 1.

A *geometric communication schedule* $\mathcal{S}$ of length T with parameters $N, \delta \in \mathbb{N}$, (N,δ)-gcs for short, is a mapping from $[N] \times [0, \delta-1]^2$ to binary sequences of length T. Let $v \in [N]$ be a station whose grid coordinates with respect to the grid G_c are equal to (i,j). We say that v *executes* (N,δ)-gcs $\mathcal{S}$ for the grid G_c in a fixed period of time, when v transmits a message in round t of that period iff the tth position of $\mathcal{S}(v, i \mod \delta, j \mod \delta)$ is equal to 1. We say that $(i_1, j_1) \equiv (i_2, j_2) \mod d$ if and only if $i_1 \equiv i_2 \mod d$ and $j_1 \equiv j_2 \mod d$. A set of stations A on the plane is δ-*diluted* wrt G_c, for $\delta \in \mathbb{N} \setminus \{0\}$, if for any two stations $v_1, v_2 \in A$ with grid coordinates (i_1, j_1) and (i_2, j_2), respectively, the relationship $(i_1, j_1) \equiv (i_2, j_2) \mod d$ holds.

Let $\mathcal{S}$ be a classical communication schedule $\mathcal{S}$ wrt N of length T, let $c > 0$ and $\delta > 0$, $\delta \in \mathbb{N}$. A δ-*dilution* of $\mathcal{S}$ wrt (N, c) is a (N,δ)-gcs (geometric communication schedule) $\mathcal{S}'$ of length $T \cdot \delta^2$ defined such such that the bit $(t-1)\delta^2 + a\delta + b$ of $\mathcal{S}'(v, a, b)$ is equal to 1 iff the bit t of $\mathcal{S}(v)$ is equal to 1. In other words, each round t of $\mathcal{S}$ is partitioned into δ^2 rounds (t, a, b) of $\mathcal{S}'$ indexed by pairs $(a, b) \in [0, \delta-1]^2$, such that a station with grid coordinates (i,j) in G_c is allowed to send messages only in rounds $(t, i \mod \delta, j \mod \delta)$, provided schedule $\mathcal{S}$ admits a transmission of this station in its round t.

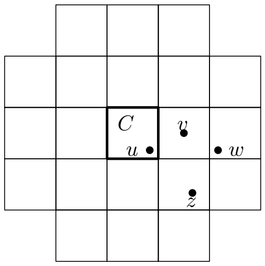


Fig. 1. If v, w, z are in the range area of u, then boxes containing v, w, and z are neighbors of C

Observe that, since ranges of stations are equal to the length of diagonal of boxes of the pivotal grid, a box $C(i,j)$ can have at most 20 neighbors (see Figure 1). We define the set DIR $\subset [-2,2]^2$ such that $(d_1,d_2) \in$ DIR iff it is possible that boxes with coordinates (i,j) and $(i+d_1, j+d_2)$ can be neighbors. For each $(d_1,d_2) \in$ DIR and $i,j \in \mathbb{Z}$, we say that a box $C' = C(i+d_1, j+d_2)$ is located in direction (d_1,d_2) from the box $C(i,j)$.

Proposition 1. *For each $\alpha > 2$ ($\alpha = 2$, resp.) there exists a constant d ($d = O(\log N)$, resp.) such that the following property is satisfied:*
Let A be a δ-diluted, wrt the pivotal grid G_γ, set of stations on the plane such that $\delta \geq d$ and each box contains at most one element of A. Then, if all elements of A transmit messages simultaneously in the same round t and no other station is transmitting, each of them transmit successfully.

The correctness of the above proposition simply follows from the fact that $\sum_{i=1}^{\infty} 1/i^{\sigma} = O(1)$ for each $\sigma > 1$, and $\sum_{i=1}^{n} 1/i = O(\log n)$ for $n \in \mathbb{N}$ (these sums appear when one evaluates maximal possible interference generated by elements of $A \setminus \{v\}$ to a box containing any neighbor of $v \in A$).

A set of stations A on the plane is *Δ-dense* if at most Δ elements of A are located in each box of the pivotal grid. A station $v \in A$ is *grid-isolated* if v is the only element of A in box(v).

Definition 1 (SINR selector). *A (N,δ)-gcs $\mathcal{S}$, for $N, \delta \in \mathbb{N}$, is a (N,δ,Δ,μ)-SINR-selector if for each Δ-dense set A with IDs in the range $[N]$, the following properties are satisfied:*
(a) *At least $\mu \cdot |A|$ stations from A transmit successfully during execution of $\mathcal{S}$ on A.*
(b) *Let $B \subseteq A$ be the set of elements of A which are not grid-isolated. Then, at least $\mu \cdot |B|$ stations from B transmit successfully during execution of $\mathcal{S}$ on A.*

Lemma 1. *For each $N \in \mathbb{N}$ and $\Delta \leq N$, there exists a $(N,\delta,\Delta,1/2)$-SINR-selector $\mathcal{S}$ of size: $O(\Delta \log^2 N)$ for $\alpha > 2$, and $O(\Delta \log^3 N)$ for $\alpha = 2$, where $\delta = O(\log N)$. Moreover, $\mathcal{S}$ is a δ-dilution of some classical communication schedule.*

The proof of Lemma 1 is based on the probabilistic method. Its idea is as follows. Consider a randomly generated classical communication schedule S, such that each $v \in [N]$ is chosen to be a transmitter in round i with probability $\frac{1}{\Delta}$. The goal is to show that S satisfies the properties of a SINR-selector after applying d-dilution with appropriately chosen d. Unfortunately, unlike in radio networks, it is not sufficient to prove that a station v is the only (successful) transmitter in the ball of radius r centered at v. Technical challenge here is to deal with interferences going from stations located in other (even distant) boxes, even though in expectation there are not many of them. We deal with this issue by bounding the probabilities that many boxes may contain more than 2^i transmitting stations, for increasing values of i, and calculating probabilities of large interferences using these bounds. Although for a particular box it might be the case that the fraction of successfully transmitting stations from this box is small, we can still prove a global bound on the number of successfully transmitting stations from the whole considered set A. Moreover, it can be shown that such a randomly generated schedule is a SINR-selector with high probability.

4 Backbone Structure and Algorithm

In this section we describe a method of building a *backbone network*, a subnetwork of an input wireless network which acts as a tool to perform several communication tasks. Given a network with a communication graph G, a *backbone* of G is a subnetwork H which satisfies the following properties:

P_1 : the stations of H form a connected dominating set of G;
P_2 : the number of stations in H is $O(m)$, where m is the size of the smallest connected dominating set of G;
P_3 : each station from $G \setminus H$ is associated with exactly one its neighbor that belongs to H (its *leader*).

Moreover, we define protocols A_1 and A_2 for a network G and its backbone H, with the following characteristics:

A_1 : a protocol simulating one round of a message passing network on H in a *multi-round*, which consists of constant number of rounds; more precisely, this protocol makes possible to exchange messages between each pair of neighbors of H in one multi-round; moreover, each message received by a station $v \in H$ is successfully broadcasted to all stations from G associated with v (i.e., stations for which v is the leader);
A_2 : a protocol which, assuming that each station v has its own message, makes possible to transmit message of each station to its leader in $O(\Delta)$ rounds.

Below, we describe efficient protocol(s) which build a backbone of the underlying network. Our algorithms strongly rely on SINR-selectors. First, leaders in boxes of the pivotal grid are chosen (Local Leader Election), then these leaders acquire knowledge about stations in their boxes and their neighbors in the communication graph (Local Leader Election and Neighborhood Learning) and finally a constant number of stations are added in each box (of the pivotal grid) to these leaders in order to form the network H satisfying stated above properties (Inter-Box Communication). Below, we describe these phases of the algorithm in more detail. We assume that Δ is known to all stations of a network. We make a simplifying assumption that, if at most one station from each box of the pivotal grid transmits in a round t, then each such transmission is successful. Due to Proposition 1, one can achieve this property using dilution with constant parameter d for $\alpha > 2$ (or $d = O(\log N)$ for $\alpha = 2$), which does not change the asymptotic complexity of our algorithm. We also use the parameter δ which corresponds to the dilution parameter from Lemma 1.

Local Leader Election. The goal of this phase is to choose the leader in each nonempty box of the pivotal grid (i.e., each box containing at least one station of G). Each station v has a local variable $\mathrm{st}(v)$ denoting its state. At the beginning $\mathrm{st}(v) =$ *active* for each station. Our algorithm consists of $\log N$ executions of $(N, \delta, \Delta, 1/2)$-SINR-selector S. However, in each step of an execution, only stations in state *active* are allowed to send messages. After each round t of the algorithm, each station v which can hear a message in t sent by a station $u \in \mathrm{box}(v)$ changes its state to *passive*. After $\log N$ executions of S, each station v such that $\mathrm{st}(v) =$active changes its state to *leader* and becomes the leader of its box.

Let $M(i)$ be the set of stations which, after the ith execution of the SINR-selector S, are in state active and they are located in boxes which contain at least two stations in state active at this time.

Proposition 2. $|M(i)| \leq |M(i-1)|/2$ *for each* $i \in [\log N]$.

Proof. Let *canonical execution* of S be an execution in which the set of stations participating during the execution does not change. The properties of S imply that the set X of stations from $M(i-1)$ which transmit successfully during the ith execution of S has at least $|M(i-1)|/2$ stations. Let $X = X_1 \cup X_2$, where X_1 contains those stations which became passive before having a successful transmission in S, and $X_2 = X \setminus X_1$. Note that v does not belong to $M(i)$ for each $v \in X_2$, since all stations in $\mathrm{box}(v)$ except of v become passive after a successful transmission of v. On the other hand, stations from X_1 does not belong to $M(i)$ either, since they are in state passive by definition. Thus, $M(i) \subseteq M(i-1) \setminus X$ and therefore $|M(i)| \leq |M(i-1)|/2$.

Lemma 2. *After* $\log N$ *application of SINR-selector* S *during* Local Leader Election, *there is exactly one station in state leader in each nonempty box.*

Proof. Observe that $|M(0)| \leq N$ and therefore, by Proposition 2, $|M(\log N)| = 0$. Thus, there is no box with more than one active station at the end of the algorithm.

It remains to verify that each nonempty box has a station in state active at the end of the algorithm. However, note that the number of active stations in a box in a particular round t may decrease only in the case that at least one station in that box is transmitting in t. Thus, this station remains active. Therefore, it is not possible that a nonempty box has no station in state active at the end of the algorithm.

Local Learning. The goal of this phase is to assure that each leader of a box gets knowledge about all stations in its box and shares this knowledge with these stations. To this aim, we execute the SINR-selector S on the set of non-leaders

$\log N$ times. And, the leaders send confirmation messages after each round of the selector if they can hear a message from their boxes. And in turn, stations which receive confirmations of their messages, get *passive*. Moreover, leaders (and other stations) store counters of confirmed stations from their boxes and arrays containing IDs of confirmed stations. Below, we describe this idea in more detail.

Algorithm 1. LOCAL LEARNING

```
1: for each station $v$: if $\mathrm{st}(v) \neq$ leader, then $\mathrm{st}(v) \leftarrow$active;
2: Repeat $\log N$ times the SINR-selector $S$. In each round, only stations in the state
   active can send messages. Each round $t$ of the selector is followed by an extra round
   $t'$, in which each station $v$ executes the following instructions $3-6$:
3:   if $\mathrm{st}(v) =$ leader, $\mathrm{box}(v) = C$ and $v$ received a message from $u \in C$ in $t$ then
4:     $count \leftarrow count + 1$
5:     $set[count] \leftarrow u$
6:     send a message containing the ID $u$, the coordinates of $u$ and $count$.
7: After the round $t'$, each station $w$ which receives a message from the leader of its
   box, updates its local copies of $count$ and $set$ appropriately.
```

The properties of S and an analysis similar to the analysis of the phase LOCAL LEADER ELECTION directly imply the following result.

Lemma 3. *After* LOCAL LEARNING *each station knows all stations located in its box, stored in its array* $set[1, count]$.

Neighborhood Learning. Note that leaders of boxes form a "backbone" which is a dominating set of the communication graph. In fact such a dominating set is at most $|\mathrm{DIR}| + 1 = 21$ times larger than a smallest dominating set (since each station v can be only in range of stations located in box $C = \mathrm{box}(v)$ and boxes in directions $(d_1, d_2) \in \mathrm{DIR}$ from C). Our goal is to add a constant number of stations from each nonempty box to the backbone, in order to guarantee connectivity of the backbone and make efficient communication inside the backbone possible. To this aim we first design a communication schedule which gives each station some knowledge about neighbors of each station in its box. This is achieved by allowing that, given a box C, each station from C sends a message in different round. Thus, each such message is received by all neighbors of the transmitting station (due to dilution). Moreover, each station adds locations and IDs of all stations from which it receives messages to (stored locally) the set of its neighbors. Given a box $C(i, j)$ and $(d_1, d_2) \in \mathrm{DIR}$, the key information for us is whether and which stations from C have neighbors in the box $C(i+d_1, j+d_2)$. Therefore, we repeat the second time the procedure in which each station in each box transmits separately. This time each station v attaches to its messages $D(v)$, the set of directions from DIR describing boxes in which v has neighbors, and for each such direction (d_1, d_2), it attaches $twin_{(d_1,d_2)}(v)$, the smallest ID of such a neighbor[3]. Thanks to that, this information is spread in the whole $\mathrm{box}(v)$.

[3] We do not allow a station to send information about all its neighbors in order to keep messages small, i.e., of size $O(\log N)$.

Given this information, stations "responsible" for communication with other boxes are chosen in each box.

Algorithm 2. Neighborhood Learning

```
1: Each station $v$ sets: $\Gamma(v) = \emptyset$
2: for each station $v$ do
3:     for $i = 1, 2, \ldots, \Delta$ do
4:         if $v = set[i]$ then
5:             $v$ sends a message,
6:         else if $v$ can hear $u$: $\Gamma(v) \leftarrow \Gamma(v) \cup \{u\}$
7: for each station $v$ do
8:     for $i = 1, 2, \ldots, \Delta$ do
9:           if $v = set[i]$ then $v$ sends $D(v)$.
10:          if $v$ can hear $D(u)$ then $v$ stores it.
11: for $(d_1, d_2) \in \mathrm{DIR}$ and $v \in C = C(i, j)$ do
12:     $C' \leftarrow C(i + d_1, j + d_2)$
13:     $nb_{(d_1,d_2)} \leftarrow \{u \in C \mid u$ has a neighbor in $C'\}$
14:     if $nb_{(d_1,d_2)} \neq \emptyset$: $s^C_{(d_1,d_2)} \leftarrow \min(nb_{(d_1,d_2)})$
15:       $r^{C'}_{(-d_1,-d_2)} \leftarrow twin_{(d_1,d_2)}(s^C_{(d_1,d_2)})$.
16:       $s^C_{(d_1,d_2)}$ sends a message to $r^{C'}_{(-d_1,-d_2)}$.
```

Note that each station can perform computation from line 10-14 independently using information received in the first two loops. All stations of type $s^C_{(d_1,d_2)}$ and $r^C_{(d_1,d_2)}$ are added to the backbone H (except of leaders of all boxes). The idea is that the task of $s^C_{(d_1,d_2)}$ (senders) is to send a message from the box C to the box C' located in direction (d_1, d_2) from C, while the task of $r^{C'}_{(-d_1,-d_2)}$ (receivers) is to broadcast this message to all stations in C'. That is, H consists of stations with status leader and stations $s^C_{(d_1,d_2)}$, $r^C_{(d_1,d_2)}$ for each $(d_1, d_2) \in \mathrm{DIR}$ and box C.

Simulation of Message Passing Model Inside the Backbone. Now, we define *multi-round* which consists of $|\mathrm{DIR}| \cdot (\delta')^2$ actual rounds of our network, where δ' corresponds to the parameter δ from Proposition 1.

Algorithm 3. Multi-round

```
1: for each station $v$ do
2:     if $\mathrm{st}(v) = $ leader then $v$ sends a message;
3: for $(d_1, d_2) \in \mathrm{DIR}$ do
4:     for each station $v$ do
5:         Round 1: if $(v = s^{\mathrm{box}(v)}_{(d_1,d_2)})$:
6:           send a mess.;
7:         Round 2: if $(v = r^{\mathrm{box}(v)}_{(-d_1,-d_2)})$:
8:           send a mess. received in Round 1;
```

Note that, since at most one station from each box is sending a message, each stations is transmitting successfully. Moreover, since each station from H

is sending during a multi-round, each station can (successfully) send a message to all its neighbors in H during a multi-round. Below, we formulate a corollary emphasizing the fact that our backbone provides possibility of communication in a graph whose vertices correspond to boxes of the pivotal grid and edges connecting boxes which are neighbors in a network (see Figure 1).

Lemma 4. *Assume that all stations in C know the same message M_C, for a box C of the pivotal grid. During a multi-round, the message M_C can be transmitted to each station $v' \in C'$, for each pair of boxes C, C' that are neighbors in the underlying network.*

Theorem 1. *The backbone H satisfying properties $P_1 - P_3$ and protocols satisfying A_1, A_2 can be build in time $O(\Delta \operatorname{polylog} N)$.*

Note that protocols A_1, A_2 can be obtained by Lemma 4 and Lemma 3, resp.

5 Applications of Backbone

Using the result of Theorem 1, one can efficiently simulate algorithms designed for message passing networks in the SINR model. Such a simulation requires $O(\Delta \operatorname{polylog} N + T\Delta)$ rounds, where T is the time complexity of the simulated algorithm in the message passing network without interference. However, if an execution of the simulated algorithm can be done only on the backbone of the underlying network, the multiplicative factor Δ disappears, i.e., time complexity of such simulation is $O(\Delta \operatorname{polylog} N + T)$, where the factor $\Delta \operatorname{polylog} N$ corresponds to the algorithm building the backbone.

An example of such a situation is the leader election problem. In order to choose the leader of a network, one can build the backbone H, choose the leader among elements of H and then, using protocol A_1, broadcast ID of the leader to the remaining stations in $O(1)$ rounds. If the diameter of a network D is known in advance, one can choose a leader by a simple flooding algorithm in $O(D)$ rounds (recall that the degree of stations in H is bounded by a constant). As we show in the full version of the paper, a similar complexity can be achieved also in the case that D is not known to the stations in advance.

Theorem 2. *There is a deterministic algorithm choosing a leader in $O(D + \Delta \operatorname{polylog} N)$ rounds, where D is the diameter of the underlying network.*

We complement the above result by a lower bound, which leaves only a polylogarithmic gap in complexity. Interestingly, it also holds for randomized algorithms. Here, we say that a randomized algorithm works in time $f(n)$, where n is the size of a network, if it accomplishes the task (on every network) in $f(n)$ rounds with probability at least $1 - \tau(n)$, where $\tau(n) = o(1)$.

Theorem 3. *Each (randomized) algorithm solving leader election problem works in $\Omega(\Delta + D)$ rounds in the worst case.*

Finally, using the backbone structure and the leader election algorithm, we can solve the multi-broadcast problem. Assume that k distinct messages are located in various stations of a network. Our algorithm uses the backbone structure and protocols A_1, A_2 in order to collect all messages in leaders of appropriate boxes. Then, the global leader of the entire network is chosen (see Theorem 2). Simultaneously, a rooted spanning tree of the backbone H is build, the diameter of this tree is proportional to the diameter of H. Next, using techniques of Cidon et al. [5], all messages are collected in the root of the tree. Finally, all messages are broadcasted by a pipelined flooding algorithm along the spanning tree. The following theorem characterizes efficiency of our solution.

Theorem 4. *There is a deterministic algorithm that finishes multi-broadcast in* $O(D + k + \Delta \,\text{polylog}\, N)$ *rounds.*

6 Final Remarks and Extensions

It is worth noting that our algorithm constructing the backbone is local in nature, and it tolerates a small constant inaccuracy in the location data. The construction of efficient SINR-selector can be complex (we only showed its existence); in practice, one could select such structure using randomness, which guarantees that with very high probability the obtained structure will be a SINR-selector.

We have not specified exact complexities of algorithms, because they depend on the actual value of the path loss parameter α. Since in our applications SINR-selectors are applied $\log N$ times, Lemma 1 implies that the $\text{polylog}\, N$ factor in our algorithms is equal to $\log^3 N$ for $\alpha > 2$ and to $\log^4 N$ for $\alpha = 2$.

References

1. Avin, C., Lotker, Z., Pasquale, F., Pignolet, Y.-A.: A Note on Uniform Power Connectivity in the SINR Model. In: Dolev, S. (ed.) ALGOSENSORS 2009. LNCS, vol. 5804, pp. 116–127. Springer, Heidelberg (2009)
2. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. J. Comput. Syst. Sci. 45(1), 104–126 (1992)
3. Censor-Hillel, K., Gilbert, S., Kuhn, F., Lynch, N.A., Newport, C.C.: Structuring unreliable radio networks. In: PODC, pp. 79–88. ACM (2011)
4. Chung, H.C., Robinson, P., Welch, J.L.: Optimal regional consecutive leader election in mobile ad-hoc networks. In: FOMC, pp. 52–61. ACM (2011)
5. Cidon, I., Kutten, S., Mansour, Y., Peleg, D.: Greedy packet scheduling. SIAM J. Comput. 24(1), 148–157 (1995)
6. Czumaj, A., Rytter, W.: Broadcasting algorithms in radio networks with unknown topology. In: FOCS, pp. 492–501. IEEE Computer Society (2003)
7. DeMarco, G.: Distributed broadcast in unknown radio networks. SIAM J. Comput. 39(6), 2162–2175 (2010)
8. Dessmark, A., Pelc, A.: Broadcasting in geometric radio networks. J. Discrete Algorithms 5(1), 187–201 (2007)

9. Emek, Y., Gasieniec, L., Kantor, E., Pelc, A., Peleg, D., Su, C.: Broadcasting in udg radio networks with unknown topology. Distributed Computing 21(5), 331–351 (2009)
10. Fanghänel, A., Kesselheim, T., Räcke, H., Vöcking, B.: Oblivious interference scheduling. In: PODC, pp. 220–229. ACM (2009)
11. Galcík, F., Gasieniec, L., Lingas, A.: Efficient broadcasting in known topology radio networks with long-range interference. In: PODC, pp. 230–239. ACM (2009)
12. Goussevskaia, O., Moscibroda, T., Wattenhofer, R.: Local broadcasting in the physical interference model. In: DIALM-POMC, pp. 35–44. ACM (2008)
13. Goussevskaia, O., Pignolet, Y.A., Wattenhofer, R.: Efficiency of wireless networks: Approximation algorithms for the physical interference model. Foundations and Trends in Networking 4(3), 313–420 (2010)
14. Hobbs, N., Wang, Y., Hua, Q.-S., Yu, D., Lau, F.C.M.: Deterministic Distributed Data Aggregation under the SINR Model. In: Agrawal, M., Cooper, S.B., Li, A. (eds.) TAMC 2012. LNCS, vol. 7287, pp. 385–399. Springer, Heidelberg (2012)
15. Jurdzinski, T., Kowalski, D.: Distributed backbone structure for deterministic algorithms in the sinr model of wireless networks. CoRR abs/1207.0602v2 (2012)
16. Kesselheim, T.: A constant-factor approximation for wireless capacity maximization with power control in the sinr model. In: SODA, pp. 1549–1559. SIAM (2011)
17. Kesselheim, T., Vöcking, B.: Distributed Contention Resolution in Wireless Networks. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 163–178. Springer, Heidelberg (2010)
18. Kowalski, D.R., Pelc, A.: Optimal deterministic broadcasting in known topology radio networks. Distributed Computing 19(3), 185–195 (2007)
19. Kowalski, D.R., Pelc, A.: Leader Election in Ad Hoc Radio Networks: A Keen Ear Helps. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part II. LNCS, vol. 5556, pp. 521–533. Springer, Heidelberg (2009)
20. Kuhn, F., Lynch, N.A., Newport, C.C.: The abstract mac layer. Distributed Computing 24(3-4), 187–206 (2011)
21. Kushilevitz, E., Mansour, Y.: An omega(d log (n/d)) lower bound for broadcast in radio networks. SIAM J. Comput. 27(3), 702–712 (1998)
22. Richa, A., Scheideler, C., Schmid, S., Zhang, J.: Towards jamming-resistant and competitive medium access in the sinr model. In: S3 2011, pp. 33–36. ACM (2011)
23. Scheideler, C., Richa, A.W., Santi, P.: An o(log n) dominating set protocol for wireless ad-hoc networks under the physical interference model. In: MobiHoc, pp. 91–100. ACM (2008)
24. Yu, D., Hua, Q.-S., Wang, Y., Tan, H., Lau, F.C.M.: Distributed Multiple-Message Broadcast in Wireless Ad-Hoc Networks under the SINR Model. In: Even, G., Halldórsson, M.M. (eds.) SIROCCO 2012. LNCS, vol. 7355, pp. 111–122. Springer, Heidelberg (2012)
25. Yu, D., Wang, Y., Hua, Q.S., Lau, F.C.M.: Distributed local broadcasting algorithms in the physical interference model. In: DCOSS, pp. 1–8. IEEE (2011)

Distributed Online and Stochastic Queuing on a Multiple Access Channel*

Marcin Bienkowski[1], Tomasz Jurdzinski[1,4], Miroslaw Korzeniowski[2,3], and Dariusz R. Kowalski[4]

[1] Institute of Computer Science, University of Wrocław, Poland
[2] Inst. of Mathematics and Computer Science, Wrocław Univ. of Technology, Poland
[3] LaBRI, Univeristy of Bordeaux 1, France
[4] Department of Computer Science, University of Liverpool, UK

Abstract. We consider the problems of online and stochastic packet queuing in a distributed system of n nodes with queues, where the communication between the nodes is done via a multiple access channel. In each round, an arbitrary number of packets can be injected into the system, each to an arbitrary node's queue. Two measures of performance are considered: the total number of packets in the system, called the total load, and the maximum queue size, called the maximum load. In the online setting, we develop a deterministic algorithm that is asymptotically optimal with respect to both complexity measures, in a competitive way. More precisely, the total load of our algorithm is bigger then the total load of any other algorithm, including centralized offline solutions, by only $O(n^2)$, while the maximum queue size of our algorithm is at most n times bigger than the maximum queue size of any other algorithm, with an extra additive $O(n)$. The optimality for both measures is justified by proving the corresponding lower bounds. Next, we show that our algorithm is stochastically optimal for *any* expected injection rate smaller or equal to 1. To the best of our knowledge, this is the first solution to the stochastic queuing problem on a multiple access channel that achieves such optimality for the (highest possible) rate equal to 1.

1 Introduction

Multiple Access Channel is one of the fundamental models for distributed communication. It has been widely studied and used in the context of theoretical analysis of Ethernet and wireless protocols, contention resolution in systems with buses, and in other emerging technologies. Roughly speaking, a multiple access channel models environments in which distributed nodes/resources compete for access to the shared communication and distribution channel, and in case of contention, no contender wins the access.

* Supported by MNiSW grant number N N206 368839, 2010-2013. The work of the second and the fourth author was supported by the Engineering and Physical Sciences Research Council [grant number EP/G023018/1]. The third author is supported by grant 2010/342643 of the Institute of Mathematics and Computers Science of the Wroclaw University of Technology.

Distributed queuing on a multiple access channel is one of the most fundamental problems, widely studied by both theoreticians (cf. [7]) and practitioners (cf. [4]). In this problem, packets arrive continuously at nodes, and the goal is to maintain bounded queues and latency, whenever possible. The main two lines of research include design and analysis of protocols in two scenarios: for restricted adversarial injection patterns and for stochastic injections. The best up-to-date results guarantee bounded queues only for *arbitrarily bounded* packet burst, in case of the former setting, and only for stochastic injection rates *smaller* than 1 in the latter one. This work aims to resolve the remaining cases of heavy traffic in affirmative. We believe that the newly developed and analyzed distributed scheduling techniques could substantially improve flow stability in heavy traffic systems, such as data and video streaming under 802.11aa.

The Model. We consider the scenario where n nodes with pairwise disjoint ids in $\{1, 2, \dots, n\}$ broadcast packets and communicate through a multiple access channel (MAC). Each node has a buffer, also called a *queue*, of potentially infinite capacity. We assume slotted time, where times are numbered from 0. At time 0, the adversary may inject arbitrary number of packets, each placed at an arbitrary node. Then, for $t = 1, 2, 3, \dots$, the following happens:

– In round t, defined as an interval between times $t-1$ and t, any node may transmit a message containing at most one packet. A transmission is successful if exactly one node transmits in the round; in such case, the transmitted packet is removed from the queue of the transmitting node.
– Then, at time t, the adversary injects arbitrary number of packets, each placed into an arbitrary node queue.

We assume Ethernet-like capabilities of MAC, i.e., each node can simultaneously listen and transmit, and thus knows whether the transmission was successful or not. However, our positive results work also in the model, where a station sending a message cannot listen at the same time. We assume that nodes can communicate only through MAC, but they are allowed to append control bits to the sent packets. We do not impose any restriction on the number of such bits, however we argue later that in a single message our algorithm appends only $O(\log n)$ additional bits of information to a transmitted packet. Note that control bits are inevitable in order to achieve competitiveness, as proved in [8] even for restricted adversaries and $n \geq 3$.

We consider two models of analysis of online queuing on a multiple access channel: competitive and stochastic. The former approach is new, in the sense that only bounded burst injection patterns have been considered so far, without comparison to the optimal solution. We describe the competitive approach in the remainder of this section. The detailed description of the stochastic queuing setting is deferred to Section 3.

Competitive Ratio. For any algorithm ALG and a packet injection pattern $\mathcal{I}$, let $Q_{\text{ALG}}(\mathcal{I}, t, i)$ denote the length of the queue (the number of pending packets) at node i at time t. Let $L_{\text{ALG}}(\mathcal{I}, t) = \sum_{i=1}^{n} Q_{\text{ALG}}(\mathcal{I}, t, i)$; we call $L_{\text{ALG}}(\mathcal{I}, t)$ the *total load* at time t under injection pattern $\mathcal{I}$. Finally, let $M_{\text{ALG}}(\mathcal{I}, t)$ be the *maximum load* under injection pattern $\mathcal{I}$, i.e., $M_{\text{ALG}}(\mathcal{I}, t) = \max_i Q_{\text{ALG}}(\mathcal{I}, t, i)$.

We call an online distributed algorithm (R, A)-competitive for minimizing the total load if for any adversarial pattern of packet injections $\mathcal{I}$ and any time step t it holds that $L_{\text{ALG}}(\mathcal{I}, t) \leq R \cdot L_{\text{OPT}}(\mathcal{I}, t) + A$ where OPT is the optimal *offline centralized* solution for injection pattern $\mathcal{I}$ until round t.

We call a randomized online distributed algorithm (R, A)-competitive for minimizing the total load if for any adversarial pattern of packet injections $\mathcal{I}$ and any time step t it holds that $\mathrm{E}[L_{\text{ALG}}(\mathcal{I}, t)] \leq R \cdot L_{\text{OPT}}(\mathcal{I}, t) + A$, where the expectation is taken over all random choices of the algorithm up to the step t.

For both deterministic and randomized algorithms we define competitiveness for minimizing maximum load in analogous way. We emphasize that the relation between ALG and OPT has to hold for *any* step t and *any* injection pattern $\mathcal{I}$. Note that, unlike in the traditional approach of competitive analysis [5], we explicitly give the additive factor in the competitive ratio.

1.1 Previous and Related Work

To the best of our knowledge, this is the first work studying online distributed queuing problem for unrestricted packet injection patterns. We analyze, in a competitive way, two important complexity measures: total load and max-load. In what follows, we describe a related work including online queuing in the centralized model and queuing under restricted adversarial injection patterns. Next we provide a summary of results for stochastic optimality of protocols, so far obtained only for injection rates smaller than 1.

Online Queuing in the Centralized Model. The optimization problems considered in this paper were also analyzed in the setting where *central coordination* is assumed and all nodes have *global knowledge* about all injected packets. Clearly, minimizing the total load is no longer a challenge in such setting, as any work-conserving algorithm (i.e., transmitting packets from non-empty queue whenever possible) is optimal with respect to the total load minimization. However, minimizing the length of the maximal queue is non-trivial and known in the literature under the name of *balanced scheduling*. In particular, Fleischer and Koga [10], and independently Bar-Noy et al. [3], proved that any algorithm serving always a longest nonempty queue achieves asymptotically optimal competitive ratio of $\Theta(\log n)$, including also randomized solutions. Fleischer and Koga [10] proved additionally that the popular round-robin algorithm is $\Omega(m)$-competitive, where m is the number of injected packets. Note that the latter result qualifies as non-competitive in case of unbounded number of injected packets. The comparison of results for the centralized model and the distributed one, obtained in this work, is given in Table 1. In particular, the discrepancy between the results in these two models shows that the lack of central coordination tremendously affects the performance of the whole system.

Online Queuing in the Distributed Setting under Restricted Adversaries. Inspired by adversarial queuing problems in store-and-forward packet

Table 1. The bounds on the competitiveness in the centralized and distributed settings, for the two complexity measures: total load and max-load

	centralized	distributed
minimizing total load	OPT (straightforward)	OPT + $\Theta(n^2)$ (this paper)
minimizing maximum load	$\Theta(\log n) \cdot$ OPT [3,10]	$n \cdot$ OPT + $O(n)$ (this paper)

networks [2,6], several papers analyzed *distributed* queuing on a multiple access channel under *restricted* adversarial injection patterns. Previous works by Chlebus et al. [8] and Anantharamu et al. [1] considered adversaries that were (ρ, b)-restricted, also called (ρ, b)-leaky-bucket, for $\rho \leq 1$ and fixed $b \geq 1$. The restriction is that in each time interval I, the adversary may only inject $\rho \cdot |I| + b$ packets into the system. Moreover, the solutions were analyzed in a worst-case manner with respect to parameters n, ρ, b. Restricted adversaries were also used for modelling jamming on a multiple access channel [14].

We emphasize that the unrestricted adversary considered in this work may not only generate all injection patterns allowed for the restricted case, but also patterns with some periods of "burstiness" growing arbitrarily large that were not allowed by the restricted adversary. Moreover, the previous results for the restricted adversary provided only global bounds on queue sizes in case they were bounded, while competitive analysis provided in this work compares the solution to the optimal algorithm at any single round. Although algorithms designed and analyzed under the restricted adversarial injection patterns may not imply similar results in more general competitive model considered in this work, some lower bounds can be adopted. In particular, Chlebus et al. [8] proved that, even in the $(1, 1)$-restricted setting, no algorithm achieves bounded latency. This implies that no algorithm is competitive with respect to the latency measure, and motivates our focus on the total and maximum load measures instead.

Stochastic Queuing. There is a rich history of research on *stochastic* queuing on a multiple access channels, i.e., when packets are injected subject to statistical constraints. See the surveys by Gallager [12] and Chlebus [7] for an overview of early and middle-stage research. In particular, Håstad et al. [13] proved stochastic optimality of polynomial backoff protocols for any fixed injection rate smaller than 1 and disproved it in case of exponential backoff. To the best of our knowledge, all the previous results concerning stochastic optimality were proved for expected fixed injection rates *strictly smaller* than 1. Ours is the first deterministic distributed online algorithm achieving stochastic optimality also for (highest possible) injection rate 1.

1.2 Our Results

We develop a deterministic distributed online algorithm SCAT, whose competitiveness is asymptotically optimal with respect to both the total number of packets in the system and the maximum queue size.

Theorem 1. *The algorithm* SCAT *is* $(1, n^2 + 4n)$*-competitive for the total load measure and* $(n, 5n)$*-competitive for the maximum load measure.*

That is, the total load of our algorithm is, in each round, larger by an additive factor of $O(n^2)$ than the total load of the best offline algorithm (taken for the same adversarial injection pattern). Moreover, the maximal queue size of our algorithm is, in each round, at most n times larger than the maximal queue size of the best offline algorithm, plus an additive factor $O(n)$. The optimality of both bounds is justified by the following results, the former holding even for randomized algorithms.

Theorem 2. *For any randomized algorithm* ALG *which is* (R, A)*-competitive for maximum queue minimization, it holds that* $R \geq n$.

Theorem 3. *For any deterministic algorithm* ALG *which is* (R, A)*-competitive for total load minimization, it holds that* $R \geq 1$ *and* $A \geq (n/2 - 1)^2 - 1$.

See Table 1 for a summary of results concerning competitiveness of distributed online queuing on a multiple-access channel versus centralized online queuing. Although one can argue that such big bounds on the optimal values of competitiveness parameters diminish importance of our model, observe that these bounds do not depend on time of an execution of a protocol (which might be arbitrarily large).

Furthermore, we show efficiency of our algorithm with respect to the *stochastic queuing problem* in a distributed setting for expected injection rate 1. More precisely, we show that our algorithm reaches the state with empty queues infinitely many times with probability 1, regardless of the initial distribution of packets, provided packets are injected to the system randomly according to the Bernoulli distribution with the expected number of 1 packet per round. (Note that rate 1 is the highest possible to obtain so defined stochastic optimality.) All previous solutions to this variant of the problem guaranteed such property only for the expected number of packets per round *strictly smaller* than 1. For this case, we show even a stronger property: that the expected number of steps needed to reach the state with empty queues is finite.

Due to space limit, omitted proofs will appear in the full version of the paper.

Distributed Online Solution: Challenges and Ideas. Our main online algorithm SCAT is designed to overcome two fundamental challenges imposed by the shared channel: delay in updating information (there is at most one node transmitting successfully at a time, therefore a common knowledge about majority of nodes come from $\Omega(n)$ rounds in the past), and waste (i.e., collision or silence) caused during information gathering or otherwise by scheduling packets without fairly accurate information.

To demonstrate these problems, consider the behavior of already studied protocols. Probably the simplest one is the round-robin protocol, in which nodes transmit (and gather information) periodically according to some pre-defined list. It generates an unbounded waste when the adversary injects all packets to a single node, one packet per round. One could modify this protocol to empty the whole queue where visiting a node, which would prevent such waste as considered before. However, an unbounded waste is obtained in a slightly more sophisticated scenario when the adversary injects one packet per round to a fixed node i until this node starts to be processed; then packets are injected to the node preceding i (again, one packet per round), and so on.

Another idea would be to use a buffer to amortize the waste generated by checking the queues of potentially empty node: such an idea was introduced by Chlebus et al. [8], who proposed algorithm Move-Big-To-Front (MBTF for short). In this algorithm, the round robin procedure is applied until a node with queue larger than n is found; in such case, the queue is moved into the beginning of the round robin list and emptied in consecutive rounds down to the level of exactly n pending packets. Then the round robin sub-routine is applied again, and the whole process is repeated in a loop. Observe however that the adversary can first fill each node to the level of at least $n/2$ packets, by injecting one packet per round on average, and then—by injecting packets always to the last node on the list—create queues of size $\Omega(n^2)$, while the optimum solution has at most one packet in the whole system at each round.

The above examples are token-based protocols. Bianchi [4] argued that randomized backoff protocols are not stable under highly saturated injection patterns. In general, as we also demonstrate in the proof of one of our lower bounds, using ad hoc transmission pattern may cause even more waste comparing to the best offline solution, as collisions may occur due to simultaneous transmissions.

Our solution introduces a specific potential function that efficiently trades a delay in obtaining information about queue sizes for the waste caused by silent rounds. More precisely, the algorithm runs in two modes: scanning and trimming. The former is to update the information, the latter is to transmit packets so to compete with the optimal solution. The potential function defines the order of scanned and trimmed nodes, and conditions when to switch between the two modes (i.e., efficiently between the progress in information update and in keeping the queues balanced).

The result in the stochastic injection setting is obtained by proving (positive) recurrence of the underlying Markov chains in two steps. First, we define and analyze some idealistic Markov chains, corresponding to the behavior of offline solutions. In particular, we prove that properties of these Markov chains imply stability of the optimal offline protocol in the stochastic injection setting. Next, by applying the competitiveness result from the worst-case online analysis, we argue that the stochastic process corresponding to the execution of our online algorithm satisfies stochastic optimality.

2 Competitive Algorithm SCAT

In this section, we show a protocol Scan-And-Trim (Scat) and prove that it is $(n, O(n))$-competitive. We start with a high-level description, accompanied by intuitions. Then, we provide the pseudo-code and a sketch of the analysis.

The number of nodes n and the id of a node are the only input parameters for the algorithm executed by the node. The protocol is *collision-avoiding*: it schedules transmissions in such a way that collisions never occur. To this end, it builds on a token-passing paradigm, in which a unique node with the “token-holder” status transmits a message. Recall that in the considered setting, a message contains at most one packet and a number of additional bits of information. In our protocol, the transmitting node attaches only the current size (i.e., the number of packets) of its queue.

We assume that if in a round the token holder has no packet to transmit, it still transmits a message, but it contains no packet, only the number zero representing its empty queue. Such a round we call *void*. Note that this is for notational simplicity, as we may assume that not transmitting anything has the same semantics.

Our algorithm abstracts from the local queuing policy (such as Fifo, Lifo, Sis, etc.) as it does not influence the considered measures of performance. In practice, Fifo queue could be seen as the most "fair" queuing policy.

Global State. There is a certain number of variables stored by the algorithm at each node. In particular, each node keeps information which node holds the token, the current mode of operation, and the list of all the nodes augmented with additional data. While the exact description of these variables is given later, we emphasize that the values of these variables are the same for all nodes. This is achieved by (i) initializing all these variables to the same values, (ii) ensuring that their evolution is deterministic and depends solely on their current value and the information transmitted in a given round. Recall that the protocol is collision free, and therefore the information heard by all nodes is the same.

Hence, we call these variables *global*, keeping in mind that, in fact, they are stored locally, but coherence between these variables' values is ensured. We also emphasize that except for its own packet queue, no node holds any other non-global information.

The most important global variable is a list $\mathcal{L}$ of nodes. It contains ids of all the nodes stored in a certain order; the positions of $\mathcal{L}$ are numbered from 1 to n. Whenever we write "node j", we mean the node with the j-th position on list $\mathcal{L}$. Additionally, $\mathcal{L}$ stores three pieces of information for each node q:

– Key, equal to the queue size of q attached to the last message transmitted by q on the channel; the queue size is computed without the tranmitted packet. If q has not yet transmitted, the key is set to zero;
– Threshold value for q, equal to a non-negative integer, whose value will be determined later (and modified only at some particular rounds of the protocol).
– Queue non-emptiness indicator equal to 1 when the queue of q was non-empty when it transmitted its last message (i.e., the round was non-void and an actual packet was sent) and 0 otherwise.

The key, the threshold, and the indicator of the i-th element on the list are denoted k_i, $\varphi(i)$, and p_i, respectively. Apart from $\mathcal{L}$, there are two global variables, described in detail in the definition of the algorithm:

– *token*, a number i from $[1, n]$ denoting that the current token holder is the i-th node from $\mathcal{L}$;
– *mode*, which can be either *scanning* or *trimming*;

The main problem the algorithm has to cope with is the information delay: the keys stored in $\mathcal{L}$ are usually outdated: the nodes do not have the information about the recent changes to the queues made by packet injections. Instead, $\mathcal{L}$ contains information about the queue sizes of a specific node from the time this node last transmitted. A great advantage of the global variables—representing only a partial knowledge of the system—is that they allow for more coordinated approach, which is easier to analyze. It appears that such approach is sufficient to achieve good performance.

Potentials. For any n non-negative integers $x_1, x_2, \ldots, x_n$ sorted in non-increasing order, we define a potential function $\pi : \{1, \ldots, n\} \to \mathbb{N} \cup \{0\}$. Function π is defined iteratively, from $i = 1$ up to $i = n$, as

$$\pi(i) = \min \left\{ x_i,\ S_i - \textstyle\sum_{j=1}^{i-1} \pi(j) \right\} \quad ,$$

where $S_i = \sum_{j=1}^{i} 2(n+1-j) = (2n+1-i) \cdot i$. In particular, $\pi(1) = \min\{x_1, 2n\}$. We also define the total potential as $\overline{\pi} = \sum_{i=1}^{n} \pi(i)$.

Fact 1. *For the potential function π of any values, it holds that*
1. $0 \leq \pi(i) \leq 2n$ *for any* $i \in [1, n]$,
2. $\pi(i) \geq \pi(i+1)$ *for any* $i \in [1, n-1]$,
3. $\sum_{j=1}^{i} \pi(i) \leq S_i$ *for any* $i \in [1, n]$.

Lemma 1. *Let π be a potential function and $0 < i_1 < \ldots < i_\ell = p$, where $0 < p, \ell \leq n$. Then, $\sum_{k=1}^{\ell} \pi(i_k) \leq S_\ell + \ell - p - z$, where $z = |\{j \leq p \,|\, \pi(j) = 0\}|$.*

Thresholds. The algorithm uses the potential function to compute thresholds. Namely, at some points of the time (defined later; these points are the same for all nodes), $\mathcal{L}$ is sorted in the non-increasing order of keys. Ties are broken according to ids, which assures that the ordering computed locally is the same for all the nodes. At this point $k_1, k_2, \ldots, k_n$ denote the values of keys and they are a non-increasing sequence. The potential π is computed on the values of keys k_i and is stored in the thresholds $\varphi(i)$, i.e., we simply set $\varphi(i) := \pi(i)$ for all $i \in [1, n]$. Threshold values $\varphi(1), \ldots, \varphi(n)$ change only at those times. At any time, the packets at node i above the threshold $\varphi(i)$ are called *overhead packets*, i.e., node i with ℓ packets has $\max\{\ell - \varphi(i), 0\}$ overhead packets and $\min\{\varphi(i), \ell\}$ non-overhead packets.

Algorithm Definition. At time 0, the algorithm initializes its global variables. Namely, $\mathcal{L}$ is populated with ids of the nodes, sorted according to the values of id. Thresholds and keys for all nodes are set to zero, *token* is set to 1, and *mode* is set to *scanning*. Some packets may be already injected at time 0 by the adversary. Then, for $t = 1, 2, 3, \ldots$, the following happens (cf. Sect. 1 with the description of the model).

– In round t, the processor whose position on $\mathcal{L}$ is equal to *token* transmits. It transmits a message containing a packet from its queue (if it has any) along with the size of its queue (computed after removing the transmitted packet from the queue). All nodes, including the transmitting one receive this message.

Algorithm 1. Update of global variables at time t

```
case $mode = scanning$
    if $\sum_{j=1}^{token}(k_j + p_j - \varphi_j) \leq token$ and $token < n$ then
        $token \leftarrow token + 1$
    else
        sort $\mathcal{L}$ in a non-increasing order of keys
        $\varphi \leftarrow$ the potential function of keys
        if there exists $i$ such that $k_i > \varphi_i$ then
            $token \leftarrow \min\{i \mid k_i > \varphi_i\}$
            $mode \leftarrow trimming$
        else
            $token \leftarrow 1$
case $mode = trimming$
    if $\sum_{j=1}^{n}(k_j - \varphi_j) > 0$ then
        if $k_{token} \leq \varphi_{token}$ then
            $token \leftarrow \min\{\ell > token \mid k_\ell > \varphi_\ell\}$
    else
        $token \leftarrow 1$
        $mode \leftarrow scanning$
```

– Round t is divided into three actions, executed w.l.o.g. in the following order:
  1. All nodes update key k_{token} on the list $\mathcal{L}$ as well as the value of the variable p_{token} on the basis of the message they heard in round t. Precisely, if the message from the transmitting processor contains a packet, p_i is set to 1, otherwise the round is void (i.e., the transmitting processor's queue is empty and its message contains only the information that its queue size is zero), both k_{token} and p_{token} are set to 0. That is, p_i indicates whether the node i had a nonempty queue when it held the *token* for the last time.
  2. The adversary injects an arbitrary number of packets to the system; they are appended to particular queues.
  3. Each node executes Algorithm 1. Depending on the mode, all nodes execute the instructions from the corresponding case.

By this description, it is straightforward, that all nodes are capable of tracking the values of global variables.

2.1 SCAT Analysis

Below, we show that the algorithm Scat is optimal with respect to both complexity measures (maximum load and total load).

Consider an execution of algorithm Scat. Its rounds can be grouped into *scanning* and *trimming* cycles in the following way. A *trimming cycle* is just a contiguous sequence of rounds in which Scat is in trimming mode. On the other hand, the contiguous sequence of rounds in which Scat is in scanning mode consists of one or more consecutive *scanning cycles*. Precisely, in the first round of the scanning cycle, the first node from $\mathcal{L}$ has the token, and the scanning cycle ends when the outer *else* branch is executed, i.e., when either $\sum_{j=1}^{token}(k_j + p_j - \varphi_j) > token$ or $token = n$. If the former condition occurs, then we call such scanning cycle *balanced*.

As described previously, all packets kept by the algorithm are classified either as overhead or non-overhead packets. Intuitively, the total value of potential (i.e., the number of non-overhead packets in the system) describes the number of packets which are already "under control" of our algorithm (the values of the potential are at most $2n$, hence if the algorithm has only non-overhead packets it would be trivially $(0, 2n)$-competitive for the maximum load measure). Thus, in the remaining part of this section, we focus on bounding the number of overhead packets. Two possible issues may occur. First, the number of overhead packets may increase rapidly, because the adversary is allowed to inject arbitrary number of packets in each round. However, in such case even Opt has these packets in its queues. Second, when Scat recomputes thresholds at the end of some scanning cycle, if a new total threshold is lower than the current one, some of the packets may change their status from non-overhead to overhead. Showing that this occurs very rarely poses the main difficulty in our analysis.[1]

Semi-potentials. By the algorithm definition, at some times the list $\mathcal{L}$ becomes sorted according to the key values, and thresholds are set to the current values of the potential function. To establish relation between the old and new values of thresholds, we want to be able to compare these potential functions. As a direct comparison might be infeasible, we introduce a helper concept: a semi-potential function.

A function ψ is a semi-potential function with respect to non-negative integers $x_1, x_2, \ldots, x_n$ if for any $1 \leq i \leq n$, the following two properties hold: (i) $0 \leq \psi(i) \leq x_i$, and (ii) the sum of any i values among $\psi(1), \ldots, \psi(n)$ is at most S_i. The total semi-potential is defined as $\overline{\psi} = \sum_{i=1}^{n} \psi(i)$.

Unlike in the definition of the potential function, we do not require that the values of x_i are sorted. Moreover, for a fixed sorted set of values, the potential function is defined uniquely, while there might be various semi-potential functions. Clearly, for sorted keys their potential function is also a semi-potential function. Moreover, it is also the "largest" possible semi-potential, as stated next.

Lemma 2. *Fix any non-negative integers $x_1, \ldots, x_n$ sorted in non-increasing order and let π be their potential. For any semi-potential ψ of these integers, $\overline{\psi} \leq \overline{\pi}$.*

[1] It can be shown that when one simplifies the used potential function by choosing $S_i = \sum_{j=1}^{i}(n+1-j)$ instead of $S_i = \sum_{j=1}^{i} 2(n+1-j)$, the adversary can create an injection pattern causing the additional $\Omega(n)$ factor in the max-load competitiveness (both in multiplicative and additive components). It demonstrates the subtlety of the chosen potential function, which is essential to control scanning and trimming processes.

Changes in the Potential. The following crucial lemma is the heart of our analysis; it shows that it is possible to control the thresholds (potentials) for balanced scanning cycles.

Lemma 3. *Consider a balanced scanning cycle C in the execution of algorithm* Scat. *Let π be the potential at the beginning of C. Then the total potential at the end of C is at least $\overline{\pi} + y$, where y is the number of void rounds during C.*

Proof. Let $k'_1, \ldots, k'_n$ be the values of keys at the beginning of the cycle C. Let $k_1, \ldots, k_n$ be the values of keys at the end of the cycle, before they are sorted by Scat. Let $p_1, \ldots, p_n$ be the queue non-emptiness indicators at the end of the cycle. Clearly, $k_i + p_i \geq k'_i$ for any i. As π is the potential at the beginning of the cycle, $\pi(i) \leq k'_i$ and therefore, π is the semi-potential for values $k_1 + p_1, \ldots, k_n + p_n$. Thus, $\overline{\pi} \leq \sum_{i=1}^{n}(k_i + p_i)$. We show that there exists a semi-potential function ψ with respect to $k_1, \ldots, k_n$, such that $\overline{\psi} = \overline{\pi} + y$. This will immediately conclude the proof as the total potential for the sorted values $k_1, \ldots, k_n$ is at least $\overline{\psi}$ by Lemma 2.

Since we consider a balanced scanning cycle and the threshold values are equal to the values of π, the following two properties hold
(P1) $\sum_{j=1}^{\ell'}(k_i + p_i - \pi(i)) \leq \ell'$ for each $\ell' < \ell$, and
(P2) $\sum_{j=1}^{\ell}(k_i + p_i - \pi(i)) > \ell$,
where ℓ is the position (on the list $\mathcal{L}$) of the last node which transmits during the considered scanning cycle. We define the function:

$$\psi(i) = \begin{cases} k_i & \text{for } i < \ell, \\ \sum_{i=1}^{\ell} \pi(i) - \sum_{i=1}^{\ell-1} k_i + y & \text{for } i = \ell, \\ \pi(i) & \text{for } i > \ell. \end{cases}$$

Observe that the relationship $\overline{\psi} = \overline{\pi} + y$ follows directly from the definition of ψ, thus it remains to show that ψ is indeed a semi-potential function for $k_1, \ldots, k_n$.

The first property of the semi-potential function states that $0 \leq \psi(i) \leq k_i$ for all i. This condition holds trivially for $i < \ell$. For $i \geq \ell$ observe that the value of key of the i-th element on the list was also k_i at the beginning of the cycle, and thus $\pi(i) \leq k_i$. Thus, it remains to verify that $0 \leq \psi(l) \leq k_l$. As $\sum_{i=1}^{\ell} p_i$ is the number of non-void rounds of C, $\sum_{i=1}^{\ell} p_i + y = \ell$. Applying this relation to the definition of $\psi(\ell)$, we obtain that

$$\psi(\ell) = \ell + \sum_{i=1}^{\ell} \pi(i) - \sum_{i=1}^{\ell-1} k_i - \sum_{i=1}^{\ell} p_i \ .$$

Thus, using Property (P2), we obtain that $\psi(\ell) = k_i + \ell + \sum_{i=1}^{\ell}(\pi(i) - k_i - p_i) > k_\ell$. Furthermore, applying Property (P1) with $\ell' = \ell - 1$,

$$\begin{aligned} \psi(l) &= \ell + \pi(\ell) - p_\ell - \sum_{j=1}^{l-1}(k_j + p_j - \pi(j)) \\ &\geq \ell + \pi(\ell) - p_\ell - (\ell - 1) \geq \pi(\ell) \geq 0 \ . \end{aligned} \tag{1}$$

The second property of the semi-potential function states that the sum of any m values among $\psi(1), \ldots, \psi(n)$ is at most S_m. To show it, we fix a set

$I = \{i_1, \dots, i_m\}$, where $1 \le i_1 < i_2 < \dots < i_m = r \le n$ and show that $\sum_{j \in I} \psi(j) \le S_m$.

Let y_r be the number of void rounds up to the position r, i.e., $y_r = r - \sum_{j=1}^{r} p_j$. We observe that

$$\textstyle\sum_{j=1}^{r} \psi(j) \le \sum_{j=1}^{r} \pi(j) + y_r \ . \tag{2}$$

Indeed, if $r \ge \ell$, $y_r = y$ and the inequality follows directly from the definition of ψ, and for $r < \ell$, we use Property (P2) obtaining $\sum_{j=1}^{r} \psi(j) = \sum_{j=1}^{r} k_j \le r + \sum_{j=1}^{r} \pi(j) - \sum_{j=1}^{r} p_j = \sum_{j=1}^{r} \pi(j) + y_r$.

Observe that if a round j is void, then $\pi(j) \le k_j + p_j = 0$. Hence, $y_r \le |\{j \le r \,|\, \pi(j) = 0\}|$, and thus by Lemma 1,

$$\sum_{j \in I} \pi(j) \le S_m + m - r - |\{j \le r \,|\, \pi(j) = 0\}| \le S_m + m - r - y_r \ . \tag{3}$$

Let $A_< = \{j \le r \,|\, \psi(j) < \pi(j)\}$ and $A_> = \{j \le r \,|\, \psi(j) > \pi(j)\}$. Observe that $\psi(j) \ge \pi(j) - 1$ for each $j \in [1, n]$ which follows from the relationship $k_j + p_j \ge \pi(j)$, the definition of ψ and (1). Therefore, $\psi(j) = \pi(j) - 1$ for any $j \in A_<$. Thus, using (2),

$$\sum_{j \in A_>} (\psi(j) - \pi(j)) = \sum_{j=1}^{r} (\psi(j) - \pi(j)) \sum_{j \in A_<} (\pi(j) - \psi(j)) \le y_r + |A_<| \ . \tag{4}$$

Finally, combining (3) with (4), we get:

$$\begin{aligned}
\textstyle\sum_{j \in I} \psi(j) &= \textstyle\sum_{j \in I} \pi(j) + \sum_{j \in I} (\psi(j) - \pi(j)) \\
&= \textstyle\sum_{j \in I} \pi(j) + \sum_{j \in I \cap A_>} (\psi(j) - \pi(j)) - \sum_{j \in I \cap A_<} (\pi(j) - \psi(j)) \\
&\le (S_m + m - r - y_r) + (y_r + |A_<|) - |I \cap A_<| \\
&\le S_m + |I| + |A_<| - |I \cap A_<| - r \le S_m \ . \qquad \square
\end{aligned}$$

Using the crucial lemma above and applying a few observations, we may compare the *current* number of overhead packets to the number of packets in queues of Opt at any round. Given a particular adversarial pattern of packet injections up to some fixed time t, we denote the (current) number of packets exceeding the total value of threshold of the algorithm Scat by ovr_t, and the number which Opt has in queues at t by opt_t. Note that we compute these values after the adversary injects packets at time t and after the algorithm computes new values of thresholds (if it does so). Note that at time 0, all thresholds are equal to zero and since the queues of Opt and Scat are equal, i.e. $\text{ovr}_0 = \text{opt}_0$.

Lemma 4. *Fix any scanning or trimming cycle C starting at time t and ending at time $t + r$. Then $ovr_{t+r} - opt_{t+r}$ is at most*

1. *$ovr_t - opt_t + n$ if C is a non-balanced scanning cycle;*
2. *$ovr_t - opt_t$ if C is a balanced scanning cycle or a trimming cycle.*

Using the lemma above, by a simple induction we obtain the following result, stating that we may control the number of overhead packets.

Lemma 5. *For any cycle C starting at time t, it holds that $ovr_t \leq opt_t + 2n$.*

Finally, we prove upper bounds on competitiveness of Scat.

Proof (of Theorem 1). Fix any time $t+r$ belonging to a cycle C starting at time t. By Lemma 5, $\text{ovr}_t \leq \text{opt}_t + 2n$. Assume that at times $t+1, t+2, \ldots, t+r$, the adversary injected in total j (overhead) packets. Hence, $\text{opt}_{t+r} \geq \text{opt}_t + j - r$. If C is a trimming cycle, then Scat transmits an overhead packet in each step, i.e., $\text{ovr}_{t+r} = \text{ovr}_t + j - r \leq \text{opt}_{t+r} + 2n$. If C is a scanning cycle, then its length is at most n, and thus even if Scat does not transmit any overhead packets, then $\text{ovr}_{t+r} \leq \text{ovr}_t + j$ and $\text{opt}_{t+r} \geq \text{opt}_t + j - n$. In this case, $\text{ovr}_{t+r} \leq \text{opt}_{t+r} + 3n$.

In either case, $\text{ovr}_{t+r} \leq \text{opt}_{t+r} + 3n$. The number of non-overhead packets is at most $S_n = n(n+1)$. Therefore, the total number of packets at time $t+r$ is at most $\text{opt}_{t+r} + n^2 + 4n$, which shows the first part of the theorem. Furthermore, the number of non-overhead packets at any node is at most $2n$ and hence the maximum load at any time is at most $\text{opt}_{t+r} + 3n + 2n = n \cdot (\text{opt}_{t+r}/n) + 5n$. As the maximum load of Opt at step $t+r$ is at least opt_{t+r}/n, the second part of the theorem follows. □

3 Stochastic Model

In this section we assume that packets are injected according to a random distribution, defined by the sequence of numbers $p_1, \ldots, p_n \in (0,1)$. In each step, for each queue independently, one packet is injected into the queue j with probability p_j and no packet is injected with probability $1 - p_j$. Our goal is to analyze the total load of *deterministic* distributed algorithms in such scenario.

Centralized Solution. First, we focus on the centralized algorithm Opt which has the full knowledge on the queue sizes and chooses an arbitrary (and exactly one) packet to be transmitted in each step, provided at least one queue is not empty. As Opt is centralized, the actual distribution of packets is unimportant, and we simply analyze its total load. We want to investigate the conditions sufficient and necessary for reducing the total load to zero, i.e., emptying all queues.

The evolution of the Opt's total load can be described by a time-homogeneous Markov chain (also denoted Opt) whose states $S_0, S_1, S_2, \ldots$ are non-negative integers corresponding to the total load at consecutive times. In particular, S_0 is the initial number of packets in all buffers. Let Y_t be the random variable denoting the number of packets injected in step t. Recall that by the definition of our process, all Y_t are identically and independently distributed, their support is the set $\{0, \ldots, n\}$ and their mean is equal to $\sum_{j=1}^{n} p_j$. The transitions between consecutive states is then defined by

$$S_t = \begin{cases} S_{t-1} + Y_t - 1 & \text{if } S_{t-1} > 0 \\ S_{t-1} + \max\{Y_t - 1, 0\} & \text{if } S_{t-1} = 0 \end{cases}$$

In the following, we restrict our attention to time-homogeneous Markov chains only. For any such Markov chain C and any two states S and S' we denote the probability that C ever reaches state S' when it starts from S by $P_C(S \to S')$ and the expected number of steps to hit S' for the first time by $E_C(S \to S')$. We are interested in the event of Opt reaching the empty queues state, and therefore we concentrate on bounding the terms $P_C(S_0 \to 0)$ and $E_{\text{Opt}}(S_0 \to 0)$. Note that the finiteness of $E_{\text{Opt}}(S_0 \to 0)$ trivially implies that $P_{\text{Opt}}(S_0 \to 0) = 1$. The goal of this section is to present tight conditions on $\sum_{j=1}^{n} p_j$ which assure that $P_{\text{Opt}}(S_0 \to 0) = 1$ or $E_{\text{Opt}}(S_0 \to 0)$ is finite.

Clearly the Markov chain Opt is irreducible as for any two states S and S' and large enough τ, there is a positive probability that Opt changes state from S to S' within τ. Furthermore, as in each step there is a positive probability that the number of packets remains the same (i.e., the state does not change), Opt is aperiodic.

Lemma 6. *Fix any starting state S_0. If $\sum_{j=1}^{n} p_j < 1$, then $E_{\text{Opt}}(S_0 \to 0)$ is finite.*

Lemma 7. *Fix any starting state S_0. If $\sum_{j=1}^{n} p_j = 1$, then $P_{\text{Opt}}(S_0 \to 0) = 1$*

By combining Lemmas 6 and 7, we immediately get the following corollary.

Corollary 1. *Fix any starting state S_0. It holds that $P_{\text{Opt}}(S_0 \to 0) = 1$, provided $\sum_{j=1}^{n} p_j \leq 1$. Furthermore, if $\sum_{j=1}^{n} p_j < 1$, then $E_{\text{Opt}}(S_0 \to 0)$ is finite.*

Distributed Solution. Now, we move our attention to on-line deterministic algorithms. Since an online algorithm is not able to achieve better performance than Opt, the properties of Opt motivate the following definition.

Definition 1. *An algorithm* Alg *is stochastically optimal when both conditions hold:*
(i) $P_{\text{Alg}}(S \to 0) = 1$ for any $S \geq 0$ if $\sum_{i=1}^{n} p_i \leq 1$,
(ii) $E_{\text{Alg}}(S \to 0)$ is finite for any $S \geq 0$ if $\sum_{i=1}^{n} p_i < 1$.

Lemma 8. *Fix any monotonic functions $f, g : \mathbb{N} \to \mathbb{N}$. Assume that a deterministic distributed on-line algorithm* Alg *is $(1, f(n))$-competitive with respect to total load. Assume that given m packets in its queues,* Alg *transmits them all in the next $g(m)$ steps, provided no packet is injected in this period. Then,* Alg *is stochastically optimal.*

Corollary 2. Scat *is stochastically optimal.*

Proof. It is sufficient to fix the functions f and g satisfying the conditions of Lemma 8. By Theorem 1, $f(n) = O(n^2)$. Now assume that Scat has m packets in its queues and no subsequent packet is injected. Note that in a trimming cycle, a packet is sent in each round. Furthermore, at least one packet is transmitted in a scanning cycle (which consists of at most n rounds). Therefore, all m packets are transmitted in at most $g(m) = O(n \cdot m)$ rounds. □

4 Conclusions and Open Problems

We studied competitiveness of deterministic distributed algorithms with respect to two important performance measures: total and maximum load. Our solution is asymptotically optimal with respect to both measures. All our competitive results regarding distributed environment can be contrasted with centralized online queuing, and the obtained picture suggests that there is no simple way of transforming centralized online algorithms into distributed solutions. We also show a transformation from the world of competitive analysis of distributed online queuing into distributed stochastic queuing. An interesting open problem is to analyze other measures of performance, e.g., related to timing or energy efficiency, in similar online distributed frameworks.

A Remark on Message Size. In the algorithm Scat, a node holding a token attaches the current number of packets in its queue to the transmitted packet. In order to reduce the number of auxiliary bits to $O(\log n)$, the minimum of $2n$ and the current size of the queue may be sent by the node holding a token.

References

1. Anantharamu, L., Chlebus, B.S., Kowalski, D.R., Rokicki, M.A.: Deterministic broadcast on multiple access channels. In: INFOCOM, pp. 146–150. IEEE (2010)
2. Andrews, M., Awerbuch, B., Fernández, A., Leighton, F.T., Liu, Z., Kleinberg, J.M.: Universal-stability results and performance bounds for greedy contention-resolution protocols. J. ACM 48(1), 39–69 (2001)
3. Bar-Noy, A., Freund, A., Landa, S., Naor, J.S.: Competitive on-line switching policies. In: ACM-SIAM SODA, pp. 525–534 (2002)
4. Bianchi, G.: Performance analysis of the IEEE 802.11 distributed coordination function. IEEE Journal on Selected Areas in Communications 18, 535–547 (2000)
5. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis. Cambridge University Press (1998)
6. Borodin, A., Kleinberg, J.M., Raghavan, P., Sudan, M., Williamson, D.P.: Adversarial queuing theory. J. ACM 48(1), 13–38 (2001)
7. Chlebus, B.: Randomized communication in radio networks. In: Pardalos, P.M., Rajasekaran, S., Reif, J.H., Rolim, J.D.P. (eds.) Handbook on Randomized Computing, vol. I, pp. 401–456. Kluwer Academic (2001)
8. Chlebus, B.S., Kowalski, D.R., Rokicki, M.A.: Maximum throughput of multiple access channels in adversarial environments. Distributed Comp. 22(2), 93–116 (2009)
9. Chung, K.L., Fuchs, W.: On the distribution of values of sums of random variables. Memoirs of the AMS 6, 1–12 (1951)
10. Fleischer, R., Koga, H.: Balanced scheduling toward loss-free packet queuing and delay fairness. Algorithmica 38, 363–376 (2004)
11. Foster, F.G.: On the stochastic matrices associated with certain queueing processes. Ann. Math Statist. 24, 355–360 (1953)
12. Gallager, R.G.: A perspective on multiaccess channels. IEEE Transactions on Information Theory 31(2), 124–142 (1985)
13. Håstad, J., Leighton, F.T., Rogoff, B.: Analysis of backoff protocols for multiple access channels. SIAM J. Comput. 25(4), 740–774 (1996)
14. Richa, A.W., Scheideler, C., Schmid, S., Zhang, J.: Competitive and fair medium access despite reactive jamming. In: ICDCS, pp. 507–516 (2011)

Fast Distributed Computation in Dynamic Networks via Random Walks*

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

[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

Abstract. The paper investigates efficient distributed computation in *dynamic* networks in which the network topology changes (arbitrarily) from round to round. Random walks are a fundamental primitive in a wide variety of network applications; the local and lightweight nature of random walks is especially useful for providing uniform and efficient solutions to distributed control of dynamic networks. Given their applicability in dynamic networks, we focus on developing fast distributed algorithms for performing random walks in such networks.

Our first contribution is a rigorous framework for design and analysis of distributed random walk algorithms in dynamic networks. We then develop a fast distributed random walk based algorithm that runs in $\tilde{O}(\sqrt{\tau\Phi})$ rounds[1] (with high probability), where τ is the *dynamic mixing time* and Φ is the *dynamic diameter* of the network respectively, and returns a sample close to a suitably defined stationary distribution of the dynamic network.

Our next contribution is a fast distributed algorithm for the fundamental problem of information dissemination (also called as *gossip*) in a dynamic network. In gossip, or more generally, k-gossip, there are k pieces of information (or tokens) that are initially present in some nodes and the problem is to disseminate the k tokens to all nodes. We present a random-walk based algorithm that runs in $\tilde{O}(\min\{n^{1/3}k^{2/3}(\tau\Phi)^{1/3}, nk\})$ rounds (with high probability). To the best of our knowledge, this is the first $o(nk)$-time fully-distributed *token forwarding* algorithm that improves over the previous-best $O(nk)$ round distributed algorithm [Kuhn et al., STOC 2010], although in an oblivious adversary model.

Keywords: Dynamic Network, Distributed Algorithm, Random walks, Random sampling, Information Dissemination, Gossip.

* Supported in part by the following research grants: Nanyang Technological University grant M58110000, Singapore Ministry of Education Academic Research Fund Tier 2 grant MOE2010-T2-2-082, and a grant from the US-Israel Binational Science Foundation.

[1] $\tilde{O}$ hides polylog n factors where n is the number of nodes in the network.

1 Introduction

Random walks play a central role in computer science spanning a wide range of areas in both theory and practice. Random walks are used as an integral subroutine in a wide variety of network applications ranging from token management and load balancing to search, routing, information propagation and gathering, network topology construction and building random spanning trees (e.g., see [11] and the references therein). They are particularly useful in providing uniform and efficient solutions to distributed control of dynamic networks [6,23]. Random walks are local and lightweight and require little index or state maintenance which make them especially attractive to self-organizing dynamic networks such as peer-to-peer, overlay, and ad hoc wireless networks. In fact, in highly dynamic networks, where the topology can change arbitrarily from round to round (as assumed in this paper), extensive distributed algorithmic techniques that have been developed for the last few decades for *static* networks (see e.g., [21,16,22]) are not readily applicable. On the other hand, we would like distributed algorithms to work correctly and terminate even in networks that keep changing continuously over time (not assuming any eventual stabilization). Random walks being so simple and very local (each subsequent step in the walk depends only on the neighbors of the current node and does not depend on the topological changes taking place elsewhere in the network) can serve as a powerful tool to design distributed algorithms for such highly dynamic networks. However, it remains a challenge to show that one can indeed use random walks to solve non-trivial distributed computation problems efficiently in such networks, with provable guarantees. Our paper is a step in this direction.

A key purpose of random walks in many of the network applications is to perform node sampling. While the sampling requirements in different applications vary, whenever a true sample is required from a random walk of certain steps, typically all applications perform the walk naively — by simply passing a token from one node to its neighbor: thus to perform a random walk of length ℓ takes time linear in ℓ. In prior work [11,12], the problem of performing random walks in time that is significantly faster, i.e., sublinear in ℓ, was studied. In [12], a fast distributed random walk algorithm was presented that ran in time sublinear in ℓ, i.e., in $\tilde{O}(\sqrt{\ell D})$ rounds (where D is the network diameter). This algorithm used only small sized messages (i.e., it assumed the standard CONGEST model of distributed computing [21]). However, a main drawback of this result is that it applied only to *static* networks. A major problem left open in [12] is whether a similar approach can be used to speed up random walks in dynamic networks.

The goals of this paper are two fold: (1) giving fast distributed algorithms for performing random walk sampling efficiently in dynamic networks, and (2) applying random walks as a key subroutine to solve non-trivial distributed computation problems in dynamic networks. Towards the first goal, we first present a rigorous framework for studying random walks in a dynamic network (cf. Section 2). (This is necessary, since it is not immediately obvious what the output of random walk sampling in a changing network means.) The main purpose of our random walk algorithm is to output a random sample close to the "stationary distribution" (defined precisely in Section 2) of the underlying dynamic network. Our random walk algorithms work under an oblivious adversary that fully controls the dynamic network topology, but does not know the random choices made by the algorithms (cf. Section 3 for a precise problem statements and results). We present a fast distributed random walk algorithm that runs in $\tilde{O}(\sqrt{\tau\Phi})$ with high probability (w.h.p.) [2], where τ is (an upper bound on) the dynamic mixing time and Φ is the dynamic diameter of the network respectively (cf. Section 5). Our algorithm uses small-sized messages only and returns a node sample that is "close" to the stationary distribution of the dynamic network (assuming the stationary distribution remains fixed even as the network changes). (The precise definitions of these terms are deferred to Section 2). We further extend our algorithm to efficiently perform and return k independent random walk samples in $\tilde{O}(\min\{\sqrt{k\tau\Phi}, k+\tau\})$ rounds (cf. Section 6). This is directly useful in the applications considered in this paper.

Towards the second goal, we present a key application of our fast random walk sampling algorithm (cf. Section 7). We present a fast distributed algorithm for the fundamental problem of *information dissemination* (also called as *gossip*) in a dynamic network. In gossip, or more generally, k-gossip, there are k pieces of information (or tokens) that are initially present in some nodes and the problem is to disseminate the k tokens to all nodes. In an n-node network, solving n-gossip allows nodes to distributively compute any computable function of their initial inputs using messages of size $O(\log n + d)$, where d is the size of the input to the single node [14]. We present a random-walk based algorithm that runs in $\tilde{O}(\min\{n^{1/3}k^{2/3}(\tau\Phi)^{1/3}, nk\})$ rounds with high probability (cf. Section 7). To the best of our knowledge, this is the first $o(nk)$-time fully-distributed *token forwarding* algorithm that improves over the previous-best $O(nk)$ round distributed algorithm [14], albeit under an oblivious adversarial model. A lower bound of $\Omega(nk/\log n)$ under the adaptive adversarial model of [14], was recently shown in [13]; hence one cannot do substantially better than the $O(nk)$ algorithm in general under an adaptive adversary.

2 Network Model and Definitions

2.1 Dynamic Networks

We study a general model to describe a dynamic network with a *fixed* set of nodes. We consider an oblivious adversary which can make *arbitrary* changes to the graph topology in every round as long as the graph is *connected*. Such a dynamic graph process (or dynamic graph, for short) is also known as an *Evolving Graph* [3]. Suppose $V = \{v_1, v_2, \ldots, v_n\}$ be the set of nodes (vertices) and $\mathcal{G} = G_1, G_2, \ldots$ be an infinite sequence of undirected (connected) graphs on V. We write $G_t = (V, E_t)$ where $E_t \in 2^{V \times V}$ is the dynamic edge set corresponding to round $t \in \mathbb{N}$. The adversary has complete control on the topology of the

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

graph at each round, however it does not know the random choices made by the algorithm. In particular, in the context of random walks, we assume that it does not know the position of the random walk in any round (however, the adversary may know the starting position).[3] Equivalently, we can assume that the adversary chooses the entire sequence $\langle G_t \rangle$ of the graph process $\mathcal{G}$ in advance before execution of the algorithm. This adversarial model has also been used in [3] in their study of random walks in dynamic networks.

We say that the dynamic graph process $\mathcal{G}$ has some property when each G_t has that property. For technical reasons, we will assume that each graph G_t is d-regular and non-bipartite. Later we will show that our results can be generalized to apply to non-regular graphs as well (albeit at the cost of a slower running time). The assumption on non-bipartiteness ensures that the mixing time is well defined, however this restriction can be removed using a standard technique: adding self-loops on each vertices (e.g., see [3]). Henceforth, we assume that the dynamic graph is a *d-regular evolving graph* unless otherwise stated (these two terms will be used interchangeably). Also we will assume that each G_t is non-bipartite (and connected).

2.2 Distributed Computing Model

We model the communication network as an n-node dynamic graph process $\mathcal{G} = G_1, G_2, \ldots$. Every node has limited initial knowledge. Specifically, assume that each node is associated with a distinct identity number (ID). (The node ids are of size $O(\log n)$.) At the beginning of the computation, each node v accepts as input its own identity number and the identity numbers of its neighbors in G_1. The node may also accept some additional inputs as specified by the problem at hand (in particular, we assume that all nodes know n). The nodes are allowed to communicate through the edges of the graph G_t in each round t. We assume that the communication occurs in synchronous *rounds*. In particular, all the nodes wake up simultaneously at the beginning of round 1, and from this point on the nodes always know the number of the current round. We will use only small-sized messages. In particular, at the beginning of each round t, each node v is allowed to send a message of size B bits (typically B is assumed to be $O(\text{polylog}\, n)$) through each edge $e = (v, u) \in E_t$ that is adjacent to v. The message will arrive to u at the end of the current round. This is a standard model of distributed computation known as the *CONGEST(B) model* [21,19] and has been attracting a lot of research attention during last two decades (e.g., see [21] and the references therein). For the sake of simplifying our analysis, we assume that $B = O(\log^2 n)$, although this is generalizable.[4]

[3] Indeed, an adaptive adversary that always knows the current position of the random walk can easily choose graphs in each step, so that the walk never really progresses to all nodes in the network.

[4] It turns out that the per-round congestion in any edge in our random walk algorithm is $O(\log^2 n)$ bits w.h.p. Hence assuming this bound for B ensures that the random walks can never be delayed due to congestion. This simplifies the correctness proof of our random walk algorithm (cf. Lemma 1).

There are several measures of efficiency of distributed algorithms, but we will focus on one of them, specifically, *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.)

2.3 Random Walks in a Dynamic Graph

Throughout, we assume the *simple random walk* in an undirected graph: In each step, the walk goes from the current node to a random neighbor, i.e., from the current node v, the probability to move in the next step to a neighbor u is $\Pr(v,u) = 1/d(v)$ for $(v,u) \in E$ and 0 otherwise ($d(v)$ is the degree of v).

A *simple random walk* on dynamic graph $\mathcal{G}$ is defined as follows: assume that at time t the walker is at node $v \in V$, and let $N(v)$ be the set of neighbors of v in G_t, then the walker goes to one of its neighbors from $N(v)$ uniformly at random.

Let $\pi_x(t)$ define the probability distribution vector reached after t steps when the initial distribution starts with probability 1 at node x. We say that the distribution $\pi_x(r)$ is stationary (or steady-state) for the graph process $\mathcal{G}$ if $\pi_x(t+1) = \pi_x(t)$ for all $t \geq r$. Let π denote the stationary distribution vector. It is known that for every (undirected) static graph G, the distribution $\pi(v) = d(v)/2m$ is stationary. In particular, for a regular graph the stationary distribution is the uniform distribution. The *mixing time* of a random walk on a static graph G is the time t taken to reach "close" to the stationary distribution of the graph. Similar to the static case, for a d-regular evolving graph, it is easy to verify that the stationary distribution is the uniform distribution. Also, for a d-regular evolving graph, the notion of *dynamic mixing time* (formally defined below) is similar to the static case and is well defined due to the monotonicity property of distributions ($||\pi_x(t+1) - \pi|| \leq ||\pi_x(t) - \pi||$, see full paper [10]).

Definition 1. *[Dynamic mixing time] Define $\tau^x(\epsilon)$ (ϵ-near mixing time for source x) is $\tau^x(\epsilon) = \min t : ||\pi_x(t) - \pi|| < \epsilon$. Note that $\pi_x(t)$ is the probability distribution on the graph G_t in the dynamic graph process $\{G_t : t \geq 1\}$ when the initial distribution ($\pi_x(1)$) starts with probability 1 at node x on G_1. Define τ^x_{mix} (mixing time for source x) $= \tau^x(1/2e)$ and $\tau_{mix} = \max_x \tau^x_{mix}$. The dynamic mixing time is upper bounded by $\tau = \max\{$mixing time of all the static graph $G_t : t \geq 1\}$. Notice that $\tau \geq \tau_{mix}$ in general.*

We show the following theorem (proof is in the full version of the paper [10]) of dynamic mixing time.

Theorem 1. *For any d-regular connected non-bipartite evolving graph $\mathcal{G}$, the dynamic mixing time of a simple random walk on $\mathcal{G}$ is bounded by $O(\frac{1}{1-\lambda}\log n)$, where λ is an upper bound of the second largest eigenvalue in absolute value of any graph in $\mathcal{G}$. Further, it is bounded by $O(n^2 \log n)$.*

Note that the dynamic mixing time is upper bounded by the worst-case mixing time of any graph in $\mathcal{G}$, which will be (henceforth) denoted by τ. Since the second eigenvalue of the transition matrix of any regular graph is bounded by $1 - 1/n^2$, this implies that τ of a d-regular evolving graph is bounded by $\tilde{O}(n^2)$. In general, the dynamic mixing time can be significantly smaller than this bound, e.g., when all graphs in $\mathcal{G}$ have λ bounded from above by a constant (i.e., they are expanders — such dynamic graphs occur in applications e.g., [2,14]), the dynamic mixing time is $O(\log n)$.

Another parameter affecting the efficiency of distributed computation in a dynamic graph is its dynamic diameter (also called flooding time, e.g., see [4,8]). The *dynamic diameter* (denoted by Φ) of an n-node dynamic graph $\mathcal{G}$ is the worst-case time (number of rounds) required to broadcast a piece of information from any given node to all n-nodes. The dynamic diameter can be much larger than the diameter (D) of any (individual) graph G_t.

3 Problem Statements and Our Results

The Single Random Walk Problem. Given a d-regular evolving graph $\mathcal{G} = (V, E_t)$ and a starting node $s \in V$, our goal is to devise a fast distributed *random walk* algorithm such that, at the end, a destination node, sampled from a τ-length walk, outputs the source node's ID (equivalenty, one can require s to output the destination node's ID), where τ is (an upper bound on) the dynamic mixing time of $\mathcal{G}$, under the assumption that $\mathcal{G}$ is modified by an oblivious adversary (cf. Section 2). Note that this distribution will be "close" to the stationary distribution of $\mathcal{G}$ (stationary distribution and τ are both well-defined — cf. Section 2). Since we are assuming a d-regular evolving graph, our goal is to sample from (or close to) the uniform distribution (which is the stationary distribution) using as few rounds as possible. Note that we would like to sample fast via random walk — this is also very important for the applications considered in this paper. On the other hand, if one had to simply get a uniform random sample, it can be accomplished by other means, e.g., it is easy to obtain it in $O(\Phi)$ rounds (by using flooding).

For clarity, observe that the following naive algorithm solves the above problem in $O(\tau)$ rounds: The walk of length τ is performed by sending a token for τ steps, picking a random neighbor in each step. Then, the destination node v of this walk outputs the ID of s. Our goal is to perform such sampling with significantly less number of rounds, i.e., in time that is sublinear in τ, in the CONGEST model, and using random walks rather than naive flooding techniques. As mentioned earlier this is needed for the applications discussed in this paper. Our result is as follows.

Theorem 2. *The algorithm* Single-Random-Walk *(cf. Section 5) solves the Single Random Walk problem in a dynamic graph and with high probability finishes in $\tilde{O}(\sqrt{\tau\Phi})$ rounds.*

The above algorithm assumes that nodes have knowledge of τ (or at least some good estimate of it). (In many applications, it is easy to have a good estimate of τ when there is knowledge of the structure of the individual graphs — e.g., each G_t is an expanders as in [2,20].) Notice that in the worst case the value of τ is $\tilde{\Theta}(n^2)$, and hence this bound can be used even if nodes have no knowledge. Therefore putting $\tau = \tilde{\Theta}(n^2)$ in the above Theorem 2, we see that our algorithm samples a node from the uniform distribution through a random walk in $\tilde{O}(n\sqrt{\Phi})$ rounds w.h.p. Our algorithm can also be generalized to work for non-regular evolving graphs also (cf. Section 5.3).

We also consider the following extension of the Single Random Walk problem, called **the k Random Walks problem**: We have k sources $s_1, s_2, ..., s_k$ (not necessarily distinct) and we want each of the k destinations to output an ID of its corresponding source, assuming that each source initiates an independent random walk of length τ. (Equivalently, one can ask each source to output the ID of its corresponding destination.) The goal is to output all the ID's in as few rounds as possible. We show that:

Theorem 3. *The algorithm* MANY-RANDOM-WALKS *(cf. Section 6) solves the k Random Walks problem in a dynamic graph and with high probability finishes in* $\tilde{O}\left(\min(\sqrt{k\tau\Phi}, k+\tau)\right)$ *rounds.*

Information Dissemination (or k-Gossip) Problem. In k-gossip, initially k different tokens are assigned to a set V of $n(\geq k)$ nodes. A node may have more than one token. The goal is to disseminate all the k tokens to all the n nodes. We present a fast distributed randomized algorithm for k-gossip in a dynamic network. Our algorithm uses MANY-RANDOM-WALKS as a key subroutine; to the best of our knowledge, this is the first sub-quadratic time fully-distributed *token forwarding* algorithm.

Theorem 4. *The algorithm* K-INFORMATION-DISSEMINATION *(cf. Algorithm 1 in Section 7) solves* k-gossip *problem in a dynamic graph with high probability in* $\tilde{O}(\min\{n^{\frac{1}{3}}k^{\frac{2}{3}}(\tau\Phi)^{\frac{1}{3}}, nk\})$ *rounds.*

4 Related Work and Technical Overview

Dynamic networks. As a step towards understanding the fundamental computational power in dynamic networks, recent studies (see e.g., [7,14,15,13] and the references therein) have investigated dynamic networks in which the network topology changes arbitrarily from round to round. In the worst-case model that was studied by Kuhn, Lynch, and Oshman [14], the communication links for each round are chosen by an online adversary, and nodes do not know who their neighbors for the current round are before they broadcast their messages. Unlike prior models on dynamic networks, the model of [14] (like ours) does not assume that the network eventually stops changing; therefore it requires that the *algorithms work correctly and terminate even in networks that change continually over time.*

The work of [3] studied the *cover time* of random walks in an evolving graph (cf. Section 2) in an oblivious adversarial model. Recently, the work of [9], studies the flooding time of *Markovian* evolving dynamic graphs, a special class of evolving graphs.

Distributed Random Walks. Our fast distributed random walk algorithms are based on previous such algorithms designed for *static* networks [11,12]. These were the first sublinear (in the length of the walk) time algorithms for performing random walks in graphs. The algorithm of [12] performed a random walk of length ℓ in $\tilde{O}(\sqrt{\ell D})$ rounds (with high probability) on an undirected network, where D is the diameter of the network. (Subsequently, the algorithm of [12] was shown to be almost time-optimal (up to polylogarithmic factors) in [18].) The general high-level idea of the above algorithm is using a few short walks in the beginning (executed in parallel) and then carefully concatenating these walks together later as necessary. A main contribution of the present work is showing that building on the approach of [12] yields speed up in random walk computations even in dynamic networks. However, there are some challenging technical issues to overcome in this extension given the continuous dynamic nature (cf. Section 5). One key technical lemma (called the *Random walk visits Lemma*) that was used to show the almost-optimal run time of $\tilde{O}(\sqrt{\ell D})$ does not directly apply to dynamic networks. In the static setting, this lemma gives a bound on the number of times any node is visited in an ℓ-length walk, for any length that is not much larger than the cover time. More precisely, the lemma states that w.h.p. any node x is visited at most $\tilde{O}(d(x)\sqrt{\ell})$ times, in an ℓ-length walk from any starting node ($d(x)$ is the degree of x). In this paper, we show that a similar bound applies to an ℓ-length random walk on any d-regular evolving graph (cf. Lemma 4). A key ingredient in the above proof is showing that a technical result due to Lyons [17] can be made to work on an evolving graph. Other recent work involving multiple random walks in *static* networks, but in different settings include Alon et. al. [1], Elsässer et. al. [5].

Information Spreading. The main application of our random walks algorithm is an improved algorithm for information spreading or gossip in dynamic networks. To the best of our knowledge, it gives the first subquadratic, fully distributed, token forwarding algorithm in dynamic networks, partially answering an open question raised in [13]. Information spreading is a fundamental primitive in networks which has been extensively studied (see e.g., [13] and the references therein). Information spreading can be used to solve other problems such as broadcasting and leader election.

This paper's focus is on *token-forwarding* algorithms, which do not manipulate tokens in any way other than storing and forwarding them. Token-forwarding algorithms are simple, often easy to implement, and typically incur low overhead. [14] showed that under their adversarial model, k-gossip can be solved by token-forwarding in $O(nk)$ rounds, but that any deterministic online token-forwarding algorithm needs $\Omega(n \log k)$ rounds. In [13], an almost matching lower bound of $\Omega(nk/\log n)$ is shown. The above lower bound indicates that one cannot obtain efficient (i.e., subquadratic) token-forwarding algorithms for gossip in the adversarial model of [14]. This motivates considering other weaker (and perhaps more realistic) models of dynamic networks.

[13] presented a polynomial-time offline *centralized* token-forwarding algorithm that solves the k-gossip problem on an n-node dynamic network in $O(\min\{nk, n\sqrt{k \log n}\})$ rounds with high probability. This is the first known *subquadratic* time token-forwarding algorithm but it is not distributed, and furthermore, the centralized algorithm needs to know the complete evolution of

the dynamic graph in advance. It was left open in [13] whether one can obtain a fully-distributed and localized algorithm that also does not know anything about how the network evolves. In this paper, we resolve this open question in the affirmative. Our algorithm runs in $\tilde{O}(\min\{n^{1/3}k^{2/3}(\tau\Phi)^{1/3}, nk\})$ rounds with high probability. This is significantly faster than the $O(nk)$-round algorithm of [14] as well as the above centralized algorithm of [13] when τ and Φ are not too large. Note that Φ is bounded by $O(n)$ and in regular graphs τ is $O(n^2)$ ($O(n^3)$ in general graphs) and so in general, our bounds cannot be better than $O(nk)$.

5 Algorithm for Single Random Walk

5.1 Description of the Algorithm

We develop an algorithm called Single-Random-Walk (for full pseudocode cf. Algorithm 1 in the full version of the paper [10]) for d-regular evolving graph ($\mathcal{G} = (V, E_t)$). The algorithm performs a random walk of length τ (the dynamic mixing time of $\mathcal{G}$ — cf. Section 2.3) in order to sample a destination from (close to) the uniform distribution on the vertex set V.

The high-level idea of the algorithm is to perform "many" short random walks in parallel and later "stitch" the short walks to get the desired walk of length τ. In particular, we perform the algorithm in two phases, as follows. For simplicity we call the messages used in Phase 1 as "coupons" and in Phase 2 as "tokens". In Phase 1, we perform d (degree of the graph) "short" (independent) random walks of length λ (to bound the running time correctly, we show later that we do short walks of length approximately λ, instead of λ) from each node v, where λ is a parameter whose value will be fixed in the analysis. This is done simply by forwarding d "coupons" having the ID of v from v (for each node v) for λ steps via random walks. In Phase 2, starting at source s, we "stitch" some of short walks prepared in Phase 1 together to form a longer walk. 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. This can be done easily as follows: In the beginning of Phase 1, each node v assigns a coupon number for each of its d coupons. At the end of Phase 1, the coupons originating at s (containing ID of s plus a coupon number) are distributed throughout the network (after Phase 1). 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 through flooding. Let C be the sampled coupon and v be the destination node of C. s then sends a "token" to v (through flooding) and s deletes 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 more than $\tau - \lambda$ steps. Then, we complete the rest of the walk by doing the naive random walk algorithm.

To understand the intuition behind this algorithm, let us analyze its running time. First, we claim that Phase 1 needs $O(\lambda)$(see Lemma 2) rounds with high probability. Recall that, in Phase 1, each node prepares d independent random walks of length λ (approximately). We start with $d = \deg(v)$ coupons from each node v at the same time, each edge in the current graph should receive two coupons in the average case. In other words, there is essentially no congestion (i.e., not too many coupons are sent through the same edge). Therefore sending out (just) d coupons from each node for λ steps will take $O(\lambda)$ rounds in expectation. This argument can be modified to show that we need $O(\lambda)$ rounds with high probability in our model. Now by the definition of dynamic diameter, flooding takes $\varPhi$ rounds. We show that sample a coupon can be done in $O(\varPhi)$ rounds (cf. Lemma 3) and it follows that Phase 2 needs $\tilde{O}(\varPhi \cdot \tau/\lambda)$ rounds. Therefore, the algorithm needs $\tilde{O}(\lambda + \varPhi \cdot \tau/\lambda)$ which is $\tilde{O}(\sqrt{\tau\varPhi})$ when we set $\lambda = \sqrt{\tau\varPhi}$.

The reason the above algorithm for Phase 2 is incomplete is that it is possible that d coupons are not enough: We might forward the token to some node v many times in Phase 2 and all coupons distributed by v in the first phase are 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 will show (in the next section) an important property of the random walk which says that a random walk of length $O(\tau)$ will visit each node v at most $\tilde{O}(\sqrt{\tau}d)$ times (cf. Lemma 4). But this bound is not enough to get the desired running time, as it does not say anything about the distribution of the connector nodes. We use the following idea to overcome it: Instead of nodes performing walks of length λ, each such walk i do a walk of length $\lambda + r_i$ where r_i is a random number in the range $[0, \lambda - 1]$. Since the random numbers are independent for each walk, each short walks are now of a random length in the range $[\lambda, 2\lambda - 1]$. This modification is needed to claim that each node will be visited as a connector only $\tilde{O}(\sqrt{\tau}d/\lambda)$ times (cf. Lemma 5). This implies that each node does not have to prepare too many short walks. It turns out that this aspect requires quite a bit more work in the dynamic setting and therefore needs new ideas and techniques. The compact pseudo code of the above algorithm including full proofs can be found in the full version of the paper [10].

5.2 Analysis

We first show the correctness of the algorithm in the following lemma and then analyze the time complexity.

Lemma 1. *The algorithm* Single-Random-Walk*, with high probability, outputs a node sample that is close to the uniform probability distribution on the vertex set V.*

Proof. (sketch) We know (from Theorem 1) that any random walk on a regular evolving graph reaches "close" to the uniform distribution at step τ regardless of any changes of the graph in each round as long as it is d-regular, non-bipartite and connected. Therefore it is sufficient to show that Single-Random-Walk finishes with a node v which is the destination of a true random walk of length τ on some appropriate dynamic graph from the source node s. We show this below in two steps.

First we show that each short walk (of length approximately λ) created in phase 1 is a true random walk on a dynamic graph sequence $G_1, G_2, \ldots, G_{\tilde{\lambda}}$ ($\tilde{\lambda}$ is some approximate value of λ). This means that in every step t, each walk moves to some random neighbor from the current node on the graph G_t and each walk is independent of others. The proof of the Lemma 2 shows that w.h.p there is at most $O(\log^2 n)$ bits congestion in any edge in any round in Phase 1. Since we consider $CONGEST(\log^2 n)$ model, at each round $O(\log^2 n)$ bits can be sent through each edge from each direction. Hence effectively there will be no delay in Phase 1 and all walks can extend their length from i to $i+1$ in one round. Clearly each walk is independent of others as every node sends messages independently in parallel. This proves that each short walk (of a random length in the range $[\lambda, 2\lambda - 1]$) is a true random walk on the graph $G_1, G_2, \ldots, G_{\tilde{\lambda}}$.

In Phase 2, we stitch short walks to get a long walk of length τ. Therefore, the τ-length random walk is not from the dynamic graph sequence $G_1, G_2, \ldots, G_\tau$; rather it is from the sequence: $G_1, G_2, \ldots, G_{\tilde{\lambda}}, G_1, G_2, \ldots, G_{\tilde{\lambda}}, \ldots$, ($\tau/\lambda$ times approximately). The stitching part is done on the graph sequence from $G_{\tilde{\lambda}+1}, G_{\tilde{\lambda}+2}$, ... onwards. This does not affect the distribution of probability on the vertex set in each step, since the graph sequence from $G_{\tilde{\lambda}+1}, G_{\tilde{\lambda}+2}, \ldots$ is used only for communication. Also note that since we define τ to be the maximum of any static graph G_t's mixing time, it clearly reaches close to the uniform distribution after τ steps of walk. in the graph sequence $G_1, G_2, \ldots, G_{\tilde{\lambda}}, G_1, G_2, \ldots, G_{\tilde{\lambda}}, \ldots$, ($\tau/\lambda$ times approximately).

Finally, when we stitch at a node v, we are sampling a coupon (short walk) uniformly at random among many coupons (and therefore, short walks starting at v) distributed by v. It is easy to see that this stitches short random walks independently and hence gives a true random walk of longer length. Thus it follows that the algorithm Single-Random-Walk returns a destination node of a τ-length random walk (starting from s) on some evolving graph. □

Time Analysis. We show the running time of algorithm Single-Random-Walk (cf. Theorem 2) using the following lemmas. The full proofs can be found in the full version of the paper [10].

Lemma 2. *Phase 1 finishes in $O(\lambda)$ rounds with high probability.*

Lemma 3. *Sample-Coupon always finishes within $O(\Phi)$ rounds.*

We note that the adversary can force the random walk to visit any particular vertex several times. Then we need many short walks from each vertex which increases the round complexity. We show the following key technical lemma (Lemma 4) that bounds the number of visits to each node in a random walk of length ℓ. In a d-regular dynamic graph, we show that no node is visited more than $\tilde{O}(\sqrt{\tau}d/\lambda)$ times as a connector node of a τ-length random walk. For this we need a technical result on random walks that bounds the number of times a node will be visited in a ℓ-length (where $\ell = O(\tau)$) random walk. Consider a simple random walk on a connected d-regular evolving graphs on n vertices. Let $N_x^t(y)$ denote the number of visits to vertex y by time t, given the walk started at vertex x. Now, consider k walks, each of length ℓ, starting from (not necessary distinct) nodes $x_1, x_2, \ldots, x_k$.

Lemma 4. (Random Walk Visits Lemma). *For any nodes* $x_1, x_2, \ldots, x_k$, $\Pr\big(\exists y \text{ s.t. } \sum_{i=1}^{k} N_\ell^{x_i}(y) \geq 32\ d\sqrt{k\ell+1}\log n + k\big) \leq 1/n$.

This lemma says that the number of visits to each node can be bounded. However, for each node, we are only interested in the case where it is used as a connector (the stitching points). The lemma below shows that the number of visits as a connector can be bounded as well; i.e., if any node appears t times in the walk, then it is likely to appear roughly t/λ times as connectors.

Lemma 5. *For any vertex v, if v appears in the walk at most t times then it appears as a connector node at most $t(\log n)^2/\lambda$ times with probability at least* $1 - 1/n^2$.

Now we are ready to proof the main result (Theorem 2) of this section.

Proof (Proof of the Theorem 2). First, we claim, using Lemma 4 and 5, that each node is used as a connector node at most $\frac{32\ d\sqrt{\tau}(\log n)^3}{\lambda}$ times with probability at least $1 - 2/n$. To see this, observe that the claim holds if each node x is visited at most $t(x) = 32\ d\sqrt{\tau+1}\log n$ times and consequently appears as a connector node at most $t(x)(\log n)^2/\lambda$ times. By Lemma 4, the first condition holds with probability at least $1 - 1/n$. By Lemma 5 and the union bound over all nodes, the second condition holds with probability at least $1 - 1/n$, provided that the first condition holds. Therefore, both conditions hold together with probability at least $1 - 2/n$ as claimed.

Now, we choose $\lambda = 32\sqrt{\tau\Phi}(\log n)^3$. By Lemma 2, Phase 1 finishes in $O(\lambda) = \tilde{O}(\sqrt{\tau\Phi})$ rounds with high probability. For Phase 2, Sample-Coupon is invoked $O(\frac{\tau}{\lambda})$ times (only when we stitch the walks) and therefore, by Lemma 3, contributes $O(\frac{\tau\Phi}{\lambda}) = \tilde{O}(\sqrt{\tau\Phi})$ rounds.

Therefore, with probability at least $1-2/n$, the rounds are $\tilde{O}(\sqrt{\tau\Phi})$ as claimed. □

5.3 Generalization to Non-regular Evolving Graphs

By using a *lazy* random walk strategy, we can generalize our results to work for a non-regular dynamic graph also. The detailed strategy and analysis can be found in Section 6.3 in the full paper [10].

6 Algorithm for *k* Random Walks

The previous section was devoted to performing a single random walk of length τ (mixing time) efficiently to sample from the stationary distribution. In many applications, one typically requires a large number of random walk samples. A larger amount of samples allows for a better estimation of the problem at hand. In this section we focus on obtaining several random walk samples. Specifically, we consider the scenario when we want to compute k independent walks each of length τ from different (not necessarily distinct) sources $s_1, s_2, \ldots, s_k$. We show that Single-Random-Walk can be extended to solve this problem. In particular, the algorithm Many-Random-Walks (for full pseudocode cf. Algorithm 2 in the full version of the paper [10]) to compute k walks is essentially repeating the Single-Random-Walk algorithm on each source with one common/shared phase, and yet through overlapping computation, completes faster than k times the previous bound. The crucial observation is that we have to do Phase 1 only once and still ensure all walks are independent. The pseudo code of the algorithm, analysis and proof of the main result is included in the full version of the paper [10] due to space limitation.

7 Application: Information Dissemination (or *k*-Gossip)

We present a fully distributed algorithm for the *k-gossip* problem in d-regular evolving graphs (for full pseudocode cf. Algorithm 1). Our distributed algorithm is based on the centralized algorithm of [13] which consists of two phases. The first phase consists of sending some f copies (the value of the parameter f will be fixed in the analysis) of each of the k tokens to a set of *random* nodes. We use algorithm Many-Random-Walks to efficiently do this. In the second phase we simply broadcast each token t from the random places to reach all the nodes. We show that if every node having a token t broadcasts it for $O(n \log n/f)$ rounds, then with high probability all the nodes will receive the token t.

We show that our proposed k-gossip algorithm finishes in $\tilde{O}(n^{1/3}k^{2/3}(\tau\Phi)^{1/3})$ rounds w.h.p. To make sure that the algorithm terminates in $O(nk)$ rounds, we run the above algorithm in parallel with the trivial algorithm (which is just broadcast each of the k tokens sequentially; clearly this will take $O(nk)$ rounds in total) and stops when one of the two algorithm stop. Thus the claimed bound in Theorem 4 holds. The formal proof is below.

```
Algorithm 1. K-Information-Dissemination
Input: An evolving graphs $\mathcal{G} : G_1, G_2, \ldots$ and $k$ token in some nodes.
Output: To disseminate $k$ tokens to all the nodes.

Phase 1: (Send $n^{\frac{2}{3}}(k/\tau\Phi)^{\frac{1}{3}}$ copies of each token to random places)
1: Every node holding token $t$, send $f = n^{\frac{2}{3}}(k/\tau\Phi)^{\frac{1}{3}}$ copies of each token to random
   nodes using algorithm Many-Random-Walks.
Phase 2: (Broadcast each token for $O(\frac{n \log n}{f})$ rounds)
1: for each token $t$ do
2:     For the next $2n \log n/f$ rounds, let all the nodes has token $t$ broadcast the token.
3: end for
```

Proof (Proof of the Theorem 4). We are running both the trivial and our proposed algorithm in parallel. Since the trivial algorithm finishes in $O(nk)$ rounds, therefore we concentrate here only on the round complexity of our proposed algorithm.

We are sending f copies of each k token to random nodes which means we are sampling kf random nodes from uniform distribution. So using the Many-Random-Walks algorithm, phase 1 takes $\tilde{O}(\sqrt{kf\tau\Phi})$ rounds.

Now fix a node v and a token t. Let S be the set of nodes which has the token t after phase 1. Since the token t is broadcast for $2n \log n/f$ rounds, there is a set S_v^t of at least $2n \log n/f$ nodes from which v is reachable within $2n \log n/f$ rounds. This is follows from the fact that at any round at least one uninformed node will be informed as the graph being always connected. It is now clear that if S intersects S_v^t, v will receive token t. The elements of the set S were sampled from the vertex set through the algorithm Many-Random-Walks which sample nodes from close to uniform distribution, not from actual uniform distribution. We can make it though very close to uniform by extending the walk length multiplied by some constant. Suppose Many-Random-Walks algorithm samples nodes with probability $1/n \pm 1/n^2$ which means each node in S is sampled with probability $1/n \pm 1/n^2$. So the probability of a single node $w \in S$ does not intersect S_v^t is at most $(1 - |S_v^t|(\frac{1}{n} \pm \frac{1}{n^2})) = (1 - \frac{2n \log n}{f} \times \frac{n \pm 1}{n^2})$. Therefore the probability of any of the f sampled node in S does not intersect S_v^t is at most $(1 - \frac{2(n\pm1) \log n}{nf})^f \leq \frac{1}{n^{2 \pm 2/n}}$. Now using union bound we can say that every node in the network receives the token t with high probability. This shows that phase 2 uses $kn \log n/f$ rounds and sends all k tokens to all the nodes with high probability. Therefore the algorithm finishes in $\tilde{O}(\sqrt{kf\tau\Phi} + kn/f)$ rounds. Now choosing $f = n^{2/3}(k/\tau\Phi)^{1/3}$ gives the bound as $\tilde{O}(n^{1/3}k^{2/3}(\tau\Phi)^{1/3})$. Hence, the k-gossip problem solves with high probability in $\tilde{O}(\min\{n^{1/3}k^{2/3}(\tau\Phi)^{1/3}, nk\})$ rounds. □

Note that the mixing time τ of a regular dynamic graph is at most $O(n^2)$ (cf. Theorem 1). Putting this in Theorem 4, yields a better bound for k-gossip problem in a regular dynamic graph.

8 Conclusion

We presented fast and fully decentralized algorithms for performing several random walks in distributed dynamic networks. Our algorithms satisfy strong round complexity guarantees and is the first work to present robust techniques for this fundamental graph primitive in dynamic graphs. We further extend the work to show how it can be used for efficient sampling and other applications such as token dissemination. Our work opens several interesting research directions. In the recent years, several fundamental graph operatives are being explored in various distributed dynamic models, and it would be interesting to explore further along these lines and obtain new approaches for identifying sparse cuts or graph partitioning, and similar spectral quantities. Finally, these algorithmic ideas may be useful building blocks in designing fully dynamic self-aware distributed graph systems. It would be interesting to additionally consider total message complexity costs for these algorithms explicitly.

References

1. Alon, N., Avin, C., Koucký, M., Kozma, G., Lotker, Z., Tuttle, M.R.: Many random walks are faster than one. In: SPAA, pp. 119–128 (2008)
2. Augustine, J., Pandurangan, G., Robinson, P., Upfal, E.: Towards robust and efficient computation in dynamic peer-to-peer networks. In: SODA (2012)
3. Avin, C., Koucký, M., Lotker, Z.: How to Explore a Fast-Changing World (Cover Time of a Simple Random Walk on Evolving Graphs). In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 121–132. Springer, Heidelberg (2008)
4. Baumann, H., Crescenzi, P., Fraigniaud, P.: Parsimonious flooding in dynamic graphs. In: PODC, pp. 260–269 (2009)
5. Berenbrink, P., Czyzowicz, J., Elsässer, R., Gąsieniec, L.: Efficient Information Exchange in the Random Phone-Call Model. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 127–138. Springer, Heidelberg (2010)
6. Bui, M., Bernard, T., Sohier, D., Bui, A.: Random Walks in Distributed Computing: A Survey. In: Böhme, T., Larios Rosillo, V.M., Unger, H., Unger, H. (eds.) IICS 2004. LNCS, vol. 3473, pp. 1–14. Springer, Heidelberg (2006)
7. Casteigts, A., Flocchini, P., Quattrociocchi, W., Santoro, N.: Time-varying graphs and dynamic networks. CoRR, abs/1012.0009 (2010)
8. Clementi, A., Macci, C., Monti, A., Pasquale, F., Silvestri, R.: Flooding time in edge-markovian dynamic graphs. In: PODC, pp. 213–222 (2008)
9. Clementi, A., Silvestri, R., Trevisan, L.: Information spreading in dynamic graphs. In: PODC (2012)
10. Das Sarma, A., Molla, A., Pandurangan, G.: Fast Distributed Computation in Dynamic Networks via Random Walks (May 2012), http://arxiv.org/abs/1205.5525
11. Das Sarma, A., Nanongkai, D., Pandurangan, G.: Fast distributed random walks. In: PODC (2009)
12. Das Sarma, A., Nanongkai, D., Pandurangan, G., Tetali, P.: Efficient distributed random walks with applications. In: PODC, pp. 201–210 (2010)
13. Dutta, C., Pandurangan, G., Rajaraman, R., Sun, Z.: Information spreading in dynamic networks. CoRR, abs/1112.0384 (2011)
14. Kuhn, F., Lynch, N., Oshman, R.: Distributed computation in dynamic networks. In: STOC (2010)
15. Kuhn, F., Oshman, R., Moses, Y.: Coordinated consensus in dynamic networks. In: PODC, pp. 1–10 (2011)
16. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers, San Mateo (1996)
17. Lyons, R.: Asymptotic enumeration of spanning trees. Combinatorics, Probability & Computing 14(4), 491–522 (2005)
18. Nanongkai, D., Das Sarma, A., Pandurangan, G.: A tight unconditional lower bound on distributed randomwalk computation. In: PODC, pp. 257–266 (2011)
19. Pandurangan, G., Khan, M.: Theory of communication networks. In: Algorithms and Theory of Computation Handbook, 2nd edn. CRC Press (2009)
20. Pandurangan, G., Raghavan, P., Upfal, E.: Building low-diameter peer-to-peer networks. In: FOCS (2001)
21. Peleg, D.: Distributed computing: a locality-sensitive approach. SIAM, Philadelphia (2000)
22. Tel, G.: Introduction to Distributed Algorithms. Cambridge University Press, UK (1994)
23. Zhong, M., Shen, K.: Random walk based node sampling in self-organizing networks. Operating Systems Review 40(3), 49–55 (2006)

Dense Subgraphs on Dynamic Networks[⋆]

Atish Das Sarma[1], Ashwin Lall[2],
Danupon Nanongkai[3], and Amitabh Trehan[4,⋆⋆]

[1] eBay Research Labs, San Jose, CA, USA
[2] Department of Mathematics and Computer Science,
Denison University, Granville, OH, USA
[3] University of Vienna, Austria, and Nanyang Technological University, Singapore
[4] Information Systems group, Faculty of Industrial Engineering and Management,
Technion - Israel Institute of Technology, Haifa, Israel - 32000

Abstract. In distributed networks, it is often useful for the nodes to be aware of dense subgraphs, e.g., such a dense subgraph could reveal dense substructures in otherwise sparse graphs (e.g. the World Wide Web or social networks); these might reveal community clusters or dense regions for possibly maintaining good communication infrastructure. In this work, we address the problem of self-awareness of nodes in a dynamic network with regards to graph density, i.e., we give distributed algorithms for maintaining dense subgraphs that the member nodes are aware of. The only knowledge that the nodes need is that of the *dynamic diameter* D, i.e., the maximum number of rounds it takes for a message to traverse the dynamic network. For our work, we consider a model where the number of nodes are fixed, but a powerful adversary can add or remove a limited number of edges from the network at each time step. The communication is by broadcast only and follows the CONGEST model. Our algorithms are continuously executed on the network, and at any time (after some initialization) each node will be aware if it is part (or not) of a particular dense subgraph. We give algorithms that $(2+\epsilon)$-approximate the *densest subgraph* and $(3+\epsilon)$-approximate the *at-least-k-densest subgraph* (for a given parameter k). Our algorithms work for a wide range of parameter values and run in $O(D \log_{1+\epsilon} n)$ time. Further, a special case of our results also gives the first fully decentralized approximation algorithms for densest and at-least-k-densest subgraph problems for static distributed graphs.

1 Introduction

Density is a very well studied graph property with a wide range of applications stemming from the fact that it is an excellent measure of the strength of interconnectivity between nodes. While several variants of graph density problems and algorithms have been explored in the classical setting, there is surprisingly

⋆ Full version available as [13] at http://arxiv.org/abs/1208.1454

⋆⋆ Supported in part by a Technion fellowship, the NetHD MAGNET consortium, and the Technion TASP center.

little work that addresses this question in the distributed computing framework. This paper focuses on decentralized algorithms for identifying dense subgraphs in dynamic networks.

Finding dense subgraphs has received a great deal of attention in graph algorithms literature because of the robustness of the property. The density of a subgraph only gradually changes when edges come and go in a network, unlike other graph properties such as connectivity that are far more sensitive to perturbation. Density measures the *strength* of a set of nodes by the graph induced on them from the overall structure. The power of density lies in locally observing the strength of *any* set of nodes, large or small, independent of the entire network.

Dense sugraphs often give key information about the network structure, its evolution and dynamics. To quote [22]:*"Dense subgraph extraction is therefore a key primitive for any in-depth study of the nature of a large graph".* Often, dense subgraphs may reveal information about community structure in otherwise sparse graphs e.g. the World Wide Web or social networks. They are good structures for studying the dynamics of a network and have been used, for example, to study link spam [22]. It is also possible to imagine a scenario where a dynamically evolving peer-to-peer network may want to route traffic through the densest parts of its network to ease congestion; thus, these subgraphs could form the basis of an efficient communication backbone (in combination with other subgraphs selected using appropriate centrality measures).

In this paper, we expand the static CONGEST model [41] and consider a dynamic setting where the graph edges may change continually. We present algorithms for approximating the (at least size k) densest subgraph in a dynamic graph model to within constant factors. Our algorithms are not only designed to compute size-constrained dense subgraphs, but also track or maintain them through time, thereby allowing the network to be aware of dense subgraphs even as the network changes. They are fully decentralized and adapt well to rapid network failures or modifications. This gives the densest subgraph problem a special status among global graph problems: while most graph problems are hard to approximate in $o(\sqrt{n})$ time even on static distributed networks of small diameters [12,38,20], the densest subgraph problem can be approximated in polylogarithmic time (in terms of n) for small D, even in dynamic networks.

We now explain our model for dynamic networks, define density objectives considered in this paper, and state our results.

Distributed Computing Model. Consider an undirected, unweighted, connected n-node graph $G = (V, E)$. Suppose that every node (vertex) hosts a processor with unbounded computational power (though our algorithms only use time and space polynomial in n at each vertex), but with only local knowledge initially. We assume that nodes have unique identifiers. The nodes may accept some additional inputs as specified by the problem at hand. The communication is synchronous, and occurs in discrete pulses, called *rounds*. Further, nodes can send messages to each of their neighbors in every round. In our model, all the nodes wake up simultaneously at the beginning of round 1. In each round each node v is allowed to send an arbitrary message subject to the bandwidth constraint of size $O(\log n)$ bits through any edge $e = (v, u)$ that is adjacent to v, and these messages will arrive at each corresponding neighbor at the end of the current round. Our model is akin to the standard model of distributed computation known as the *CONGEST model* [41]. The message size constraint of CONGEST is very important for large-scale resource-constrained dynamic networks where running time is crucial.

Edge-Dynamic Network Model. We use the edge deletion/addition model; i.e., we consider a sequence of (undirected) graphs $G_0, G_1, \dots$ on n nodes, where, for any t, G_t denotes the state of the dynamic network $G(V, E)$ at time t, where the adversary deletes and/or inserts upto r edges at each step, i.e., $E(G_{t+1}) = (E(G_t) \setminus E_U) \cup E_V$, where $E_U \subseteq E(G_t)$ and $E_V \subseteq E(\overline{G_t})$, $|E_U| + |E_V| \leq r$ (where $\overline{G_t}$ is the complement graph of G_t). The edge change rate is denoted by the parameter r.

Following the notion in [33], we define the *dynamic diameter* of the dynamic network $G(V, E)$, denoted by D, to be the maximum time a message needs to traverse the network at any time. More formally, dynamic diameter is defined as follows:

Definition 1 (Dynamic Diameter (Adapted from [33], Definition 3)). *We say that the dynamic network $G = (V, E)$ has a dynamic diameter of D upto time t if D is the smallest positive integer such that, for all $t' \leq t$ and $u, v \in V$, we have $(u, max\{0, t' - D\}) \rightsquigarrow (v, t')$, where, for each pair of vertices x, y and times $t_1 \leq t_2$, $(x, t_1) \rightsquigarrow (y, t_2)$ means that at time t_2 node y can receive direct information, through a chain of messages, originating from node x at time t_1.*

Note that the nodes do not need to know the exact dynamic diameter D but only a (loose) approximation to it. For simplicity, we assume henceforth that the nodes know the exact value of D.

There are several measures of efficiency of distributed algorithms, but we will concentrate on one of them, specifically, *the running time*, that is, 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.)

We are interested in algorithms that can compute and maintain an approximate (at-least-k) densest subgraph of the network at all times, after a short initialization time. We say that an algorithm can compute and maintain a solution P in time T if it can compute the solution in T rounds and can maintain a solution at all times after time T, even as the network changes dynamically.

1.1 Problem Definition

Let $G = (V, E)$ be an undirected graph and $S \subseteq$ V be a set of nodes. Let us define the following:

Graph Density. The density of a graph $G(V, E)$ is defined as $|E|/|V|$.

```
Each node of $G_0$ is a processor.
Each processor starts with a list of its neighbors in $G_0$.
Pre-processing: Processors may exchange messages with their neighbors.
for $t := 1$ to $T$ do
    Adversary deletes and/or inserts upto $r$ edges at each step i.e. $E(G_{t+1}) = (E(G_t) \setminus E_U) \cup E_V$, where $E_U \subseteq E(G_t)$ and $E_V \subseteq E(\overline{G_t})$ (where $\overline{G_t}$ is the complement graph of $G_t$).
    if edge $(u, v)$ is inserted or edge $(u, v)$ is deleted then
        Nodes $u$ and $v$ may update their information and exchange messages with their neighbors.
        Computation phase:
        Nodes may communicate (synchronously, in parallel) with their immediate neighbors. These messages are never lost or corrupted, may contain the names of other vertices, and are received by the end of this phase.
    end if
    At the end of this phase, we call the graph $G_t$.
end for
```

Success metrics:

1. **Approximate Dense Subgraphs:** *Graph S'_T:* The induced graph of a set $S'_T \subseteq V_T$, s.t., $\rho(S'_T) \geq \frac{\rho(S^*_T)}{\alpha}$, where $S^*_T \subseteq V$, s.t., $\rho(S^*_T) = \max \rho(S_T)$ over all $S_T \subseteq V_T$.
2. **Approximate at-least-k-Dense Subgraphs:** *Graph S^k_T:* The induced graph of a set $S^k \subseteq V, |S^k| \geq k$, s.t., $\rho(S^k) \geq \frac{\rho(S^{k*})}{\alpha}$, where $S^{k*} \subseteq V, |S^{k*}| \geq k$, s.t., $\rho(S^{k*}) = \max \rho(S)$ over all $S \subseteq V, |S| \geq k$.
3. **Communication per edge.** The maximum number of bits sent across a single edge in a single recovery round. $O(\log n)$ in CONGEST model.
4. **Computation time.** The maximum total time (rounds) for all nodes to compute their density estimations starting from scratch assuming it takes a message no more than 1 time unit to traverse any edge and we have unlimited local computational power at each node.

Fig. 1. The distributed Edge Insert and Delete Model

SubGraph Density. The density of a subgraph defined by a subset of nodes S of $V(G)$ is defined as the density of the induced subgraph. We will use $\rho(S)$ to denote the density of the subgraph induced by S. Therefore, $\rho(S) = \frac{|E(S)|}{|S|}$. Here $E(S)$ is the subset of edges (u, v) of E where $u \in S$ and $v \in S$. In particular, when talking about the density of a subgraph defined by a set of vertices S induced on G, we use the notation $\rho_G(S)$. We also use $\rho_t(S)$ to denote $\rho_{G_t}(S)$. When clear from context, we omit the subscript G.

The problem we address in this paper is to construct distributed algorithms to discover the following:

– **(Approximate) Densest subgraphs:** The densest subgraph problem is to find a set $S^* \subseteq V$, s.t. $\rho(S^*) = \max \rho(S)$ over all $S \subseteq V$. A α-approximate solution S' will be a set $S' \subseteq V$, s.t. $\rho(S') \geq \frac{\rho(S^*)}{\alpha}$.
– **(Approximate) At-least-k-densest subgraphs:** The densest at-least-k-subgraph problem is the previous problem restricted to sets of size at least k, i.e., to find a set $S^{k*} \subseteq V, |S^{k*}| \geq k$, s.t. $\rho(S^{k*}) = \max \rho(S)$ over all

$S \subseteq V, |S| \geq k$. A α-approximate solution S^k will be a set $S^k \subseteq V, |S^k| \geq k$, s.t. $\rho(S^k) \geq \frac{\rho(S^{k*})}{\alpha}$.

In the distributed setting, we require that every node knows whether it is in the solution S' or S^k or not. We note that the latter problem is NP-Complete, and thus it is crucial to consider approximation algorithms. The former problem can be solved *exactly* in polynomial time in the centralized setting, and it is an interesting open problem whether there is an exact distributed algorithm that runs in $O(D \operatorname{poly} \log n)$ time, even in static networks.

1.2 Our Results

We give approximation algorithms for the densest and at-least-k-densest subgraph problems which are efficient even on dynamic distributed networks. In particular, we develop an algorithm that, for a fixed constant c and any $\epsilon > 0$, $(2+\epsilon)$-approximates the densest subgraph in $O(D \log_{1+\epsilon} n)$ time provided that the densest subgraph has high density, i.e., it has a density at least $(cDr \log n)/\epsilon$ (recall that r and D are the change rate and dynamic diameter of dynamic networks, respectively). We also develop a $(3+\epsilon)$-approximation algorithm for the at-least-k-densest subgraph problem with the same running time, provided that the value of the density of the at-least-k-densest subgraph is at least $(cDr \log n)/k\epsilon$. We state these theorems in a simplified form and some corollaries below. Below, ϵ can be set as any arbitrarily small constant. We note again that at the end of our algorithms, every node knows whether they are in the returned subgraph or not.

Theorem 2. *There exists a distributed algorithm that for any dynamic graph with dynamic diameter D and parameter r returns a subgraph at time t such that, w.h.p., the density of the returned subgraph is a $(2+\epsilon)$-approximation to the density of the densest subgraph at time t if the densest subgraph has density at least $\Omega(Dr \log n)$.*

Theorem 3. *There exists a distributed algorithm that for any dynamic graph with dynamic diameter D and parameter r returns a subgraph of size at least k at time t such that, w.h.p., the density of the returned subgraph is a $(3+\epsilon)$-approximation to the density of the densest at least k subgraph at time t if the densest at least k subgraph has density at least $\Omega(Dr \log n/k)$.*

We mention two special cases of these theorems informally below. We prove the most general theorem statements depending on the parameters r and D in Section 3.

Corollary 4. *Given a dynamic graph with dynamic diameter $O(\log n)$ and a rate of change $r = O(\log^{\alpha} n)$ for some constant α (i.e. r is poly-logarithmic in n), there is a distributed algorithm that at any time t can return, w.h.p., a $(2+\epsilon)$-approximation of densest subgraph at time t if the densest subgraph has density at time t at least $\Omega(\log^{\alpha+2} n)$.*

Corollary 5. *Given a dynamic graph with dynamic diameter $O(\log n)$ and a rate of change $r = O(\log^{\alpha} n)$ for some constant α (i.e. r is poly-logarithmic in n), there is a distributed algorithm that at any time t can return, w.h.p., a $(3+\epsilon)$-approximation of k-densest subgraph at time t if the k-densest subgraph has density at time t at least $\Omega(\log^{\alpha+2} n/k)$.*

Our algorithms follow the main ideas of centralized approximation algorithms [29,3,10]. These centralized algorithms cannot be efficiently implemented even on static distributed networks. We show how some ideas of these algorithms can be turned into time-efficient distributed algorithms with a small increase in the approximation guarantees. Similar ideas have been independently discovered and used to obtain efficient streaming and MapReduce algorithms by Bahmani et al. [7].

Notice that this is already a wide range of parameter values for which our results are interesting, since the density of densest subgraphs can be as large as $\Omega(n)$ while the diameter in peer-to-peer networks is typically $O(\log n)$, and the parameter r depends on the stability of the network. A caveat, though, is that in the theorems above, D refers to the flooding time of the dynamic network, and not the diameter of any specific snapshot - understanding a relationship between these quantities remains open.

Further, our general theorems also imply the following for static graphs (by simply setting $r = 0$). No such results were known in the distributed setting even for static graphs.

Corollary 6. *In a static graph, there is a distributed algorithm that obtains, w.h.p., $(2+\epsilon)$-approximation to the densest subgraph problem in $O(D\log n)$ rounds of the CONGEST model.*

Corollary 7. *In a static graph, there is a distributed algorithm that obtains, w.h.p, $(3+\epsilon)$-approximation to the k-densest subgraph problem in $O(D\log n)$ rounds of the CONGEST model.*

Notice that this is an unconditional guarantee for static graphs (i.e. does not require any bound on the density of the optimal) and is the first distributed algorithm for these problems in the CONGEST model.

Back to dynamic graphs, in addition to computing the $(2+\epsilon)$-approximated densest and $(3+\epsilon)$-approximated at-least-k-densest subgraphs, our algorithm can also *maintain* them *at all times* with high probability. This means that, at all times (except for a short initialization period), all nodes are aware of whether they are part of the approximated at-least-k densest subgraphs, for all k.

Even though we assume that all the nodes know the value D, all our algorithms work if some upper-bound D' of D is known instead; all the algorithms and analysis work identically using D' rather than D.

Organization. Our algorithms are described in Section 2 and the approximation guarantees are proved in Section 3. We mention related work at the end of the paper in Section 4.

2 Algorithm

2.1 Main Algorithm

The nature of our algorithm is such that we *continuously* maintain an approximation to the densest subgraph in the dynamic network. At any time, after a short initialization period, any node knows whether it is a member of the output subgraph of our algorithm. In this section, we give the description of the algorithm and fully specify the behavior of each of the nodes in the network. The running time analysis and the approximation guarantees are deferred to the following sections.

Our main protocol for maintaining a dense subgraph is given in Algorithm 1. It maintains a family of $p = O(\log_{1+\epsilon} n)$ candidates for the densest subgraph $\mathcal{F} = \{V_0, V_1, \ldots, V_p\}$, where $V_0 = V(G)$, $V_i \subseteq V_{i-1}$ for all i, along with an approximation of the number of nodes and edges in each graph $\mathcal{R} = \{(m_0, n_0), \ldots, (m_p, n_p)\}$, where each m_i and n_i are the approximate number of edges and nodes, respectively, of the subgraph of G_t (the current graph) induced by V_i. The algorithm works in phases in which it estimates the size of the current subgraph V_j and the number of edges in it using the algorithms discussed in the following subsection. At the end of the phase it computes the next subgraph V_{j+1} using a criterion in Line 9 of Algorithm 1 (explained further in Section 3). After p such rounds, the algorithm has all the information it needs to output an approximation to the densest subgraph. This process is repeated continuously, and the solution is computed from the last complete family of graphs (i.e., complete computation of p subgraphs).

At any time, the densest subgraph can be computed using the steps outlined in Algorithm 2. This procedure works simply by picking the subgraph with the highest density, even if the size of this subgraph is less than k. If the graph turns out to be less than size k, we pad it by having the rest of the nodes run a distributed procedure to elect appropriately many nodes to add to the subgraph and get its size up to at least k.

Any time a densest subgraph query is initiated in the network, the nodes simply run Algorithm 2 based on the subgraphs continuously being maintained by Algorithm 1, and compute which of them are in the approximation solution. At the end of this query, each node is aware of whether it is in the approximate densest subgraph or not.

2.2 Approximating the Number of Nodes and Edges

Our algorithms make use of an operation in which the number of nodes and edges in a given subgraph need to be computed. We just mention the algorithm idea here and present the detailed algorithm in the full version [13].

Algorithm APPROX-SIZE-ESTIMATION. We achieve this in $O(D)$ rounds using a modified version of an algorithm from [32]. Their algorithm allows for approximate counting of the size of a dynamic network with high probability. We modify it to work for any subgraph that we are interested in. We also show how it can

```
Input: $1 \geq \epsilon > 0$
Output: The algorithm maintains a family of sets of nodes $\mathcal{F} = \{V_0, V_1, \ldots, V_p\}$ and induced graph sizes $\mathcal{R} = \{(m_0, n_0), (m_1, n_1), \ldots, (m_p, n_p)\}$.
 1: Let $\delta = \epsilon/24$.
 2: Let $j = 0$. Let $V_0 = V$ (i.e., we mark every node as in $V_0$).
 3: repeat
 4:   Compute $n_j$, a $(1+\delta)$-approximation of $|V_j|$ (i.e., $(1+\delta)|V_j| \geq n_j \geq (1-\delta)|V_j|$).
      At the end of this step every node knows $n_j$. See the full version [13] for detailed
      implementation.
 5:   if $n_j = 0$ then
 6:     Let $j = 0$. (Note that we do not recompute $n_0$.)
 7:   end if
 8:   Let $G_t$ be the network at the beginning of this step. Let $H_t$ be the subgraph of
      $G_t$ induced by $V_j$. We compute $m_j$, the $(1+\delta)$-approximation of the number of
      edges in $H_t$ (i.e., $(1+\delta)|E(H_t)| \geq m_j \geq (1-\delta)|E(H_t)|$). At the end of this step
      every node knows $m_j$. See the full version [13] for detailed implementation.
 9:   Let $G_{t'}$ be the network at the beginning of this step. Let $H_{t'}$ be the subgraph
      of $G_{t'}$ induced by $V_j$. Let $V_{j+1}$ be the set of nodes in $V_j$ whose degree in $H_{t'}$ is
      at least $(1+\delta)m_j/n_j$. At the end of this step, every node knows whether it is in
      $V_{j+1}$ or not.
10:   Let $j = j + 1$.
11: until forever
```

Algorithm 1. MAINTAIN(ϵ)

be used to approximate the number of edges in this subgraph at a given time. In the interest of space, these results can be found in the full version [13] described under algorithms RANDOMIXEDAPPROXIMATECOUNTING, COUNT NODES, and COUNT EDGES.

3 Analysis

We analyze approximation ratios of the algorithm presented in Section 2, the guarantee depending on parameters of the algorithm. We divide the analysis into two parts: the first part is for the densest subgraph problem and the second for the at-least-k densest subgraph problem. Although the second part subsumes the first part (if we ignore the value of constant approximation ratio), we present the first part since it has a simpler idea and a better approximation ratio.

3.1 Analysis for the Densest Subgraph Problem

Theorem 8. *Let t be the time Algorithm 2 finishes, V_i be the output of the algorithm, H^* be the optimal solution and T be the time of one round of Algorithm 1 and 2 (i.e., $T = cD\log_{1+\epsilon} n$ for some constant c). If $\rho_t(H^*) \geq 24Tr/\epsilon$ then Algorithm 2 gives, w.h.p., a $(2+\epsilon)$-approximation, i.e.,*

$$\rho_t(V_i) \geq \rho_t(H^*)/(2+\epsilon)\,.$$

Input: k, the parameter for the densest at-least-k subgraph problem, the algorithm MAINTAIN(ϵ) (cf. Algorithm 1), and its parameter notations.
Output: The algorithm outputs a set of nodes $V_i \cup \hat{V}$ (every node knows whether it is in the set or not) such that $|V_i \cup \hat{V}| \geq k$.

```
1: Let $i = \max_i m_i / \max(k, n_i)$.
2: if $n_i < (1+\delta)k$ then
3:    Let $\Delta = (1+\delta)k - n_i$. (Every node can compute $\Delta$ locally.)
4:    repeat
5:       Every node not in $V_i$ locally flips a coin which is head with probability $\Delta/n_0$.

6:       Let $\hat{V}$ be the set of nodes whose coins return heads.
7:       Approximately count the number of nodes in $\hat{V}$ using the algorithm APPROX-
         SIZE-ESTIMATION discussed in Section 2.2 with error parameter $\delta$ passed to
         COUNT EDGES under it. Let $\Delta'$ be the result returned. (Note that $\Delta'/(1+\delta) \leq$
         $|\hat{V}| \leq (1+\delta)\Delta'$ w.h.p.)
8:    until $(1+\delta)\Delta \leq \Delta' \leq (1+2\delta)\Delta$
9: end if
10: return $V_i \cup \hat{V}$
```

Algorithm 2. DENSEST SUBGRAPH(k)

The rest of this subsection is devoted to proving the above theorem. Let t, V_i and H^* be as in the theorem statement (note that $\hat{V}$ in Algorithm 2 is empty when $k = 0$). Let t' be the time that V_i is last computed by Algorithm 1. Let t'' be the time Algorithm 1 starts counting the number of edges in V_i. We prove the theorem using the following lemmas. The main idea is to first lower bound $\rho_{t''}(V_i)$ using $\rho_{t'}(H^*)$ and then use it to obtain a lower bound for $\rho_{t'}(V_i)$ in terms of $\rho_t(H^*)$. Finally, the proof is completed by lower bounding $\rho_t(V_i)$ in terms of $\rho_{t'}(V_i)$.

Lemma 9. $\rho_{t''}(V_i) > \frac{1-\delta}{2(1+\delta)^2}\rho_{t'}(H^*)$.

Proof. Let H' be the densest subgraph of $G_{t'}$. Note that

$$\rho_{t'}(H^*) \leq \rho_{t'}(H')\,. \tag{1}$$

Let i^* be the smallest index such that $V(H') \subseteq V_{i^*}$ and $V(H') \not\subseteq V_{i^*+1}$. Note that i^* exists since the algorithm repeats until we get $V_j = \emptyset$. Let v be any vertex in $V(H') \setminus V_{i^*}$. Let $H_{t',i}$ be the subgraph of $G_{t'}$ induced by nodes in V_i. Note that

$$\rho_{t'}(H') \leq 2\deg_{H'}(v) \leq 2\deg_{H_{t',i}}(v)\,. \tag{2}$$

The first inequality is because we can otherwise remove v from H' and get a subgraph of $G_{t'}$ that has a higher density than H'. The second inequality is because $H' \subseteq H_{t',i}$. Since v is removed from V_{i^*},

$$\deg_{H_{t',i}}(v) < (1+\delta)\frac{m_{i^*}}{n_{i^*}}, \tag{3}$$

where $\delta = \epsilon/24$ as in Algorithm 1. By the definition of V_i,

$$\frac{m_{i^*}}{n_{i^*}} \leq \frac{m_i}{n_i}\,. \tag{4}$$

Note that $t - t'' \leq T$ by the definition of T. Note also that $n_i \geq (1-\delta)|V_i|$ and $m_i \leq (1+\delta)|E_{t''}(V_i)|$ with high probability. It follows that

$$\frac{m_i}{n_i} \leq \frac{1+\delta}{1-\delta}\rho_{t''}(V_i)\,. \tag{5}$$

Combining Eq.(1)-(5), we get $\rho_{t'}(H^*) < 2\frac{(1+\delta)^2}{1-\delta}\rho_{t''}(V_i)$ and thus the lemma.

We now make the following observation:

Observation 10. $\rho_{t'}(H^*) \geq (1-\delta)\rho_t(H^*)\,.$

Proof. Note that $t - t' \leq T$ and thus $E_t(H^*) - E_{t'}(H^*) \leq Tr$. Since $\rho_t(H^*) \geq Tr/\delta$, $\rho_{t'}(H^*) \geq \frac{\rho_t(H^*)\cdot|V(H^*)|-Tr}{|V(H^*)|} \geq \rho_t(H^*) - Tr > (1-\delta)\rho_t(H^*)\,.$

We now combine the above Lemma 9 and Observation 10 to obtain the following lemma:

Lemma 11. $\rho_{t'}(V_i) > (\frac{(1-\delta)^2}{2(1+\delta)^2} - \delta)\rho_t(H^*)\,.$

Proof. By directly combining Lemma 9 and Observation 10 we get the following:

$$\rho_{t''}(V_i) > \frac{(1-\delta)^2}{2(1+\delta)^2}\rho_t(H^*) \geq \frac{(1-\delta)^2}{2(1+\delta)^2\delta}Tr\,.$$

Moreover, observe that there are at most Tr edges removed from V_i in total, i.e., $E_{t''}(V_i) - E_t(\ V_i) \leq Tr$. Thus

$$\begin{aligned}\rho_{t'}(V_i) &\geq \frac{\rho_{t''}(V_i)\cdot|V_i| - Tr}{|V_i|} \geq \rho_{t''}(V_i) - Tr > \left(1 - \frac{2(1+\delta)^2\delta}{(1-\delta)^2}\right)\rho_{t''}(V_i) \\ &> \left(1 - \frac{2(1+\delta)^2\delta}{(1-\delta)^2}\right)\left(\frac{(1-\delta)^2}{2(1+\delta)^2}\rho_t(H^*)\right) = \left(\frac{(1-\delta)^2}{2(1+\delta)^2} - \delta\right)\rho_t(H^*)\,.\end{aligned}$$

We are now ready to prove the theorem.

Proof (Proof of Theorem 8). Note that $t-t' \leq T$ and thus $E_{t'}(V_i) - E_t(V_i) \leq Tr$. Note that $\rho_{t'}(V_i) > \beta\rho_t(H^*) \geq \beta Tr/\delta$, where $\beta = \frac{(1-\delta)^2}{2(1+\delta)^2} - \delta$. We have

$$\rho_t(V_i) \geq \frac{\rho_{t'}(V_i)\cdot|V_i| - Tr}{|V_i|} \geq \rho_{t'}(V_i) - Tr > (1 - \frac{\delta}{\beta})\rho_{t'}(V_i).$$

Now using Lemma 11 and the value of β, we get the following:

$$\rho_t(V_i) > (1-\frac{\delta}{\beta})\beta\rho_t(H^*) = (\beta - \delta)\rho_t(H^*) = \left(\frac{(1-\delta)^2}{2(1+\delta)^2} - 2\delta\right)\rho_t(H^*).$$

The theorem follows by observing that $\frac{(1-\delta)^2}{2(1+\delta)^2} - 2\delta \geq \frac{1}{2+\epsilon}$ for any $\epsilon \leq 1$ and $\delta \geq \epsilon/24$.

3.2 Analysis for the At-Least-k Densest Subgraph Problem

Theorem 12. *Let t be the time Algorithm 2 finishes, $V_i \cup \hat{V}$ be the output of the algorithm, H^* be the optimal solution and T be the time of one iteration of Algorithm 1 and Algorithm 2 (so $T = O(D \log_{1+\epsilon} n)$). If $k\rho_t(H^*) \geq 24Tr/\epsilon$ then Algorithm 2 returns a set $V_i \cup \hat{V}$ of size at least k that is, w.h.p., a $(3+\epsilon)$-approximated solution, i.e.,*

$$\rho_t(V_i \cup \hat{V}) \geq \rho_t(H^*)/(3+\epsilon)\,.$$

The proof of this theorem is in the full version [13], and we just mention the main idea here. The proof follows a similar framework as that of Theorem 8.

Let t, V_i and H^* be as in the theorem statement. Let t' be the time that V_i is last computed by Algorithm 1. Let t'' be the time Algorithm 1 starts counting the number of edges in V_i. The crucial difference here is to obtain a strong lower bound for $\rho_{t''}(V_i \cup \hat{V})$ in terms of $\rho_{t'}(H^*)$ and $\rho_t(H^*)$. This is then translated to a lower bound on $\rho_{t'}(V_i \cup \hat{V})$ and subsequently $\rho_t(V_i \cup \hat{V})$ to complete the proof. The crucial lemma and its proof turn out to be more involved than that of the densest subgraph theorem and the case-based analysis is detailed in the full version [13].

3.3 Running Time Analysis

In this section we analyze the time that it takes for the nodes to generate an approximation to the densest subgraph. Algorithm 1 continuously runs this procedure so that it always maintains an approximation that is guaranteed to be near-optimal since we assume that the network does not change too quickly. The time that it takes for Algorithm 1 to compute a complete family of subgraphs is simply $O(Dp) = O(D \log_{1+\epsilon} n)$ since there are $p = O(\log_{1+\epsilon} n)$ rounds (Section 2.1), each of which is completed in $O(D)$ time (Section 2.2). Note that step 9 of Algorithm 1 can be done in a single round since every node already knows m_j/n_j and can easily check, in one round, the number of neighbors in $G_{t'}$ that are in V_j.

When the nodes need to compute an approximation to the at-least-k-densest subgraph in Algorithm 2, they can do so by choosing the densest subgraph among the last complete family of subgraphs found by Algorithm 1. Unfortunately, there is no guarantee that the densest such graph has at least k nodes in it, so we fix this via padding. The subgraph is padded to contain at least k nodes by having each node that is not part of the subgraph attempt to join the subgraph with an appropriate probability. It can be shown via Chernoff bounds that, with high probability, within $O(\log n)$ such attempts there are enough nodes added to the subgraph to get its size to at least k. As a result, Algorithm 2 runs in $O(D \log n)$ time.

4 Related Work

The problem of finding size-bounded densest subgraphs has been studied extensively in the classical setting. Finding a maximum density subgraph in an undirected graph can be solved in polynomial time [23,35]. However, the problem becomes NP-hard when a size restriction is enforced. In particular, finding a maximum density subgraph of size exactly k is NP-hard [5,19] and no approximation scheme exists under a reasonable complexity assumption [28]. Recently Bhaskara et al. [9] showed integrality gaps for SDP relaxations of this problem. Khuller and Saha [29] considered the problem of finding densest subgraphs with size restrictions and showed that these are NP-hard. Khuller and Saha [29] and also Andersen and Chellapilla [3] gave constant factor approximation algorithms. Some of our algorithms are based on of those presented in [29].

Our work differs from the above mentioned ones in that we address the issues in a dynamic setting, i.e., where edges of the network change over time. Dynamic network topology and fault tolerance have always been core concerns of distributed computing [6,36]. There are many models and a large volume of work in this area. A notable recent model is the dynamic graph model introduced by Kuhn, Lynch and Oshman in [32]. They introduced a stability property called T-interval connectivity (for $T \geq 1$) which stipulates the existence of a stable connected spanning subgraph for every T rounds. Though our models are not fully comparable (we allow our networks to get temporarily disconnected as long as messages eventually make their way through it), the graphs generated by our model are similar to theirs except for our limited rate of churn. They show that they can determine the size of the network in $O(n^2)$ rounds and also give a method for approximate counting. We differ in that our bounds are sublinear in n (when D is small) and we maintain our dense graphs at all times.

We work under the well-studied CONGEST model (see, e.g., [41] and the references therein). Because of its realistic communication restrictions, there has been much research done in this model (e.g., see [36,41,39]). In particular, there has been much work done in designing very fast distributed approximation algorithms (that are even faster at the cost of producing sub-optimal solutions) for many fundamental problems (see, e.g., [17,16,26,27]). Among many graph problems studied, the densest subgraph problem falls into the "global problem" category where it seems that one needs at least $\Omega(D)$ rounds to compute or approximate (since one needs to at least know the number of nodes in the graph in order to compute the density). While most results we are aware of in this category were shown to have a lower bound of $\Omega(\sqrt{n/\log n})$, even on graphs with small diameter (see [12] and references therein), the densest subgraph problem is one example for which this lower bound does not hold.

Our algorithm requires certain size estimation algorithms as a subroutine. An important tool that also addresses network size estimation is a *Controller*. Controllers were introduced in [1] and they were implemented on 'growing' trees, but this was later extended to a more general dynamic model [30,18]. Network size estimation itself is a fundamental problem in the distributed setting and closely related to other problems like leader election. For anonymous networks and under some reasonable assumptions, exact size estimation was shown to be impossible [11] as was leader election [4] (using symmetry concerns). Since then, many probabilistic estimation techniques have been proposed using exponential and geometric distributions [32,2,37]. Of course, the problem is even more challenging in the dynamic setting.

Self-* systems [8,14,15,31,34,42,21,40,24,25,43] are worth mentioning here. Often, a crucial condition for such systems is the initial detection of a particular state. In this respect, our algorithm can be viewed as a self-aware algorithm where the nodes monitor their state with respect to the environment, and this could be used for developing powerful self-* algorithms.

5 Future Work and Conclusions

We have presented efficient decentralized algorithms for finding dense subgraphs in distributed dynamic networks. Our algorithms not only show how to compute size-constrained dense subgraphs with provable approximation guarantees, but also show how these can be *maintained* over time. While there has been significant research on several variants of the dense subgraph computation problem in the classical setting, to the best of our knowledge this is the first formal treatment of this problem for a distributed peer-to-peer network model.

Several directions for future research result naturally out of our work. The first specific question is whether our algorithms and analyses can be improved to guarantee $O(D + \log n)$ rounds instead of $O(D \log n)$, even in static networks. Alternatively, can one show a lower bound of $\Omega(D \log n)$ in static networks? Bounding the value D in terms of the instantaneous graphs and change rate r would also be an interesting direction of future work. It is also interesting to show whether the densest subgraph problem can be solved *exactly* in $O(D \operatorname{poly} \log n)$ or not in the static setting, and to develop dynamic algorithms without density lower bound assumptions. Another open problem (suggested to us by David Peleg) that seems to be much harder is the *at-most-k densest subgraph problem.* One could also consider various other definitions of density and study distributed algorithms for them, as well as explore whether any of these techniques extend directly or indirectly to specific applications. Finally, it would be interesting to extend our results from the edge alteration model to allow node alterations as well.

References

1. Afek, Y., Awerbuch, B., Plotkin, S.A., Saks, M.E.: Local management of a global resource in a communication network. In: FOCS, pp. 347–357. IEEE Computer Society (1987)
2. Aggarwal, S., Kutten, S.: Time Optimal Self-Stabilizing Spanning Tree Algorithms. In: Shyamasundar, R.K. (ed.) FSTTCS 1993. LNCS, vol. 761, pp. 400–410. Springer, Heidelberg (1993)
3. Andersen, R., Chellapilla, K.: Finding Dense Subgraphs with Size Bounds. In: Avrachenkov, K., Donato, D., Litvak, N. (eds.) WAW 2009. LNCS, vol. 5427, pp. 25–37. Springer, Heidelberg (2009)

4. Angluin, D.: Local and global properties in networks of processors (extended abstract). In: Miller, R.E., Ginsburg, S., Burkhard, W.A., Lipton, R.J. (eds.) STOC, pp. 82–93. ACM (1980)
5. Asahiro, Y., Hassin, R., Iwama, K.: Complexity of finding dense subgraphs. Discrete Appl. Math. 121(1-3), 15–26 (2002)
6. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations and Advanced Topics. John Wiley & Sons (2004)
7. Bahmani, B., Kumar, R., Vassilvitskii, S.: Densest subgraph in streaming and mapreduce. PVLDB 5(5), 454–465 (2012)
8. Berns, A., Ghosh, S.: Dissecting self-* properties. In: International Conference on Self-Adaptive and Self-Organizing Systems, pp. 10–19 (2009)
9. Bhaskara, A., Charikar, M., Vijayaraghavan, A., Guruswami, V., Zhou, Y.: Polynomial integrality gaps for strong sdp relaxations of densest k-subgraph. In: SODA, pp. 388–405 (2012)
10. Charikar, M.: Greedy Approximation Algorithms for Finding Dense Components in a Graph. In: Jansen, K., Khuller, S. (eds.) APPROX 2000. LNCS, vol. 1913, pp. 84–95. Springer, Heidelberg (2000)
11. Cidon, I., Shavitt, Y.: Message terminating algorithms for anonymous rings of unknown size. Inf. Process. Lett. 54(2), 111–119 (1995)
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: STOC, pp. 363–372 (2011)
13. Das Sarma, A., Lall, A., Nanongkai, D., Trehan, A.: Dense subgraphs on dynamic networks. CoRR abs/1208.1454 (2012)
14. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Commun. ACM 17(11), 643–644 (1974), http://dx.doi.org/10.1145/361179.361202
15. Dolev, S.: Self-stabilization. MIT Press, Cambridge (2000)
16. Dubhashi, D.P., Grandioni, F., Panconesi, A.: Distributed Algorithms via LP Duality and Randomization. In: Handbook of Approximation Algorithms and Metaheuristics. Chapman and Hall/CRC (2007)
17. Elkin, M.: An overview of distributed approximation. ACM SIGACT News Distributed Computing Column 35(4), 40–57 (2004)
18. Emek, Y., Korman, A.: New Bounds for the Controller Problem. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 22–34. Springer, Heidelberg (2009)
19. Feige, U., Kortsarz, G., Peleg, D.: The dense k-subgraph problem. Algorithmica 29 (1999)
20. Frischknecht, S., Holzer, S., Wattenhofer, R.: Networks cannot compute their diameter in sublinear time. In: SODA, pp. 1150–1162 (2012)
21. Ghosh, D., Sharman, R., Raghav Rao, H., Upadhyaya, S.: Self-healing systems - survey and synthesis. Decis. Support Syst. 42(4), 2164–2185 (2007)
22. Gibson, D., Kumar, R., Tomkins, A.: Discovering large dense subgraphs in massive graphs. In: Böhm, K., Jensen, C.S., Haas, L.M., Kersten, M.L., Larson, P.Å., Ooi, B.C. (eds.) VLDB, pp. 721–732. ACM (2005)
23. Goldberg, A.V.: Finding a maximum density subgraph. Tech. Rep. UCB/CSD-84-171, EECS Department, University of California, Berkeley (1984)
24. Hayes, T., Saia, J., Trehan, A.: The forgiving graph: a distributed data structure for low stretch under adversarial attack. Distributed Computing, 1–18, http://dx.doi.org/10.1007/s00446-012-0160-1, 10.1007, doi:10.1007/s00446-012-0160-1

25. Hayes, T.P., Saia, J., Trehan, A.: The forgiving graph: a distributed data structure for low stretch under adversarial attack. In: PODC 2009: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, pp. 121–130. ACM, New York (2009)
26. Khan, M., Pandurangan, G.: A fast distributed approximation algorithm for minimum spanning trees. Distributed Computing 20, 391–402 (2008)
27. Khan, M., Kuhn, F., Malkhi, D., Pandurangan, G., Talwar, K.: Efficient distributed approximation algorithms via probabilistic tree embeddings. In: PODC, pp. 263–272 (2008)
28. Khot, S.: Ruling out PTAS for graph min-bisection, dense k-subgraph, and bipartite clique. SIAM J. Computing 36(4), 1025–1071 (2006)
29. Khuller, S., Saha, B.: On Finding Dense Subgraphs. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part I. LNCS, vol. 5555, pp. 597–608. Springer, Heidelberg (2009)
30. Korman, A., Kutten, S.: Controller and estimator for dynamic networks. In: Gupta, I., Wattenhofer, R. (eds.) PODC, pp. 175–184. ACM (2007)
31. Korman, A., Kutten, S., Masuzawa, T.: Fast and compact self stabilizing verification, computation, and fault detection of an MST. In: Gavoille, C., Fraigniaud, P. (eds.) PODC, pp. 311–320. ACM (2011)
32. Kuhn, F., Lynch, N.A., Oshman, R.: Distributed computation in dynamic networks. In: STOC, pp. 513–522 (2010)
33. Kuhn, F., Oshman, R., Moses, Y.: Coordinated consensus in dynamic networks. In: PODC, pp. 1–10 (2011)
34. Kuhn, F., Schmid, S., Wattenhofer, R.: A Self-repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn. In: van Renesse, R. (ed.) IPTPS 2005. LNCS, vol. 3640, pp. 13–23. Springer, Heidelberg (2005)
35. Lawler, E.: Combinatorial optimization - networks and matroids. Holt, Rinehart, and Winston (1976)
36. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers, San Mateo (1996)
37. Matias, Y., Afek, Y.: Simple and Efficient Election Algorithms for Anonymous Networks. In: Bermond, J.-C., Raynal, M. (eds.) WDAG 1989. LNCS, vol. 392, pp. 183–194. Springer, Heidelberg (1989)
38. Nanongkai, D., Das Sarma, A., Pandurangan, G.: A tight unconditional lower bound on distributed randomwalk computation. In: PODC, pp. 257–266 (2011)
39. Pandurangan, G., Khan, M.: Theory of communication networks. In: Algorithms and Theory of Computation Handbook, 2nd edn. CRC Press (2009)
40. Pandurangan, G., Trehan, A.: Xheal: localized self-healing using expanders. In: Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC 2011, pp. 301–310. ACM (2011), http://doi.acm.org/10.1145/1993806.1993865
41. Peleg, D.: Distributed computing: a locality-sensitive approach. Society for Industrial and Applied Mathematics, Philadelphia (2000)
42. Poor, R., Bowman, C., Auburn, C.B.: Self-healing networks. Queue 1, 52–59 (2003), http://doi.acm.org/10.1145/846057.864027
43. Trehan, A.: Algorithms for self-healing networks. Dissertation, University of New Mexico (2010)

Lower Bounds on Information Dissemination in Dynamic Networks

Bernhard Haeupler[1] and Fabian Kuhn[2]

[1] Computer Science and Artificial Intelligence Lab, MIT, USA
haeupler@mit.edu
[2] Dept. of Computer Science, University of Freiburg, Germany
kuhn@cs.uni-freiburg.de

Abstract. We study lower bounds on information dissemination in adversarial dynamic networks. Initially, k pieces of information (henceforth called tokens) are distributed among n nodes. The tokens need to be broadcast to all nodes through a synchronous network in which the topology can change arbitrarily from round to round provided that some connectivity requirements are satisfied.

If the network is guaranteed to be connected in every round and each node can broadcast a single token per round to its neighbors, there is a simple token dissemination algorithm that manages to deliver all k tokens to all the nodes in $O(nk)$ rounds. Interestingly, in a recent paper, Dutta et al. proved an almost matching $\Omega(n + nk/\log n)$ lower bound for deterministic token-forwarding algorithms that are not allowed to combine, split, or change tokens in any way. In the present paper, we extend this bound in different ways.

If nodes are allowed to forward $b \leq k$ tokens instead of only one token in every round, a straight-forward extension of the $O(nk)$ algorithm disseminates all k tokens in time $O(nk/b)$. We show that for any randomized token-forwarding algorithm, $\Omega(n + nk/(b^2 \log n \log\log n))$ rounds are necessary. If nodes can only send a single token per round, but we are guaranteed that the network graph is c-vertex connected in every round, we show a lower bound of $\Omega(nk/(c\log^{3/2} n))$, which almost matches the currently best $O(nk/c)$ upper bound. Further, if the network is T-interval connected, a notion that captures connection stability over time, we prove that $\Omega(n + nk/(T^2 \log n))$ rounds are needed. The best known upper bound in this case manages to solve the problem in $O(n + nk/T)$ rounds. Finally, we show that even if each node only needs to obtain a δ-fraction of all the tokens for some $\delta \in [0, 1]$, $\Omega(nk\delta^3/\log n)$ are still required.

1 Introduction

The growing abundance of (mobile) computation and communication devices creates a rich potential for novel distributed systems and applications. Unlike classical networks, often the resulting networks and applications are characterized by a high level of churn and, especially in the case of mobile devices, a potentially constantly changing topology. Traditionally, changes in a network have been studied as faults or as exceptional events that have to be tolerated and possibly repaired. However, particularly in mobile applications, dynamic networks are a typical case and distributed algorithms have to properly work even under the assumption that the topology is constantly changing.

Consequently, in the last few years, there has been an increasing interest in distributed algorithms that run in dynamic systems. Specifically, a number of recent papers investigate the complexity of solving fundamental distributed computations and information dissemination tasks in dynamic networks, e.g., [2–5, 8, 9, 19, 11, 16–18]. Particularly important in the context of this paper is the synchronous, adversarial dynamic network model defined in [16]. While the network consists of a fixed set of participants V, the topology can change arbitrarily from round to round, subject to the restriction that the network of each round needs to be connected or satisfy some stronger connectivity requirement.

We study lower bounds on the problem of disseminating a bunch of tokens (messages) to all the nodes in a dynamic network as defined in [16].[1] Initially k tokens are placed at some nodes in the network. Time is divided into synchronous rounds, the network graph of every round is connected, and in every round, each node can broadcast one token to all its neighbors. If in addition, all nodes know the size of the network n, we can use the following basic protocol to broadcast all k tokens to all the nodes. The tokens are broadcast one after the other such that for each token during $n - 1$ rounds, every node that knows about the token forwards it. Because in each round, there has to be an edge between the nodes knowing the token and the nodes not knowing it, at least one new node receives the token in every round and thus, after $n - 1$ rounds, all nodes know the token. Assuming that only one token can be broadcast in a single message, the algorithm requires $k(n - 1)$ rounds to disseminate all k tokens to all the nodes.

Even though the described approach seems almost trivial, as long as we do not consider protocols based on network coding, $O(nk)$ is the best upper bound known.[2] In [16], a *token-forwarding algorithm* is defined as an algorithm that needs to forward tokens as they are and is not allowed to combine or change tokens in any way. Note that the algorithm above is a token-forwarding algorithm. In a recent paper, Dutta et al. show that for deterministic token-forwarding algorithms, the described simple strategy indeed cannot be significantly improved by showing a lower bound of $\Omega(nk/\log n)$ rounds [9]. Their lower bound is based on the following observation. Assume that initially, every node receives every token for free with probability $1/2$ (independently for all nodes and tokens). Now, with high probability, whatever tokens the nodes decide to broadcast in the next round, the adversary can always find a graph in which new tokens are learned across at most $O(\log n)$ edges. Hence, in each round, at most $O(\log n)$ tokens are learned. Because also after randomly assigning tokens with probability $1/2$, overall still roughly $nk/2$ tokens are missing, the lower bound follows. We extend the lower bound from [9] in various natural directions. Specifically, we make the contributions listed in the following. All our lower bounds hold for deterministic algorithms and for randomized algorithms assuming a strongly adaptive adversary (cf. Section 3). Our results are also summarized in Table 1 which is discussed in Section 2.

Multiple Tokens Per Round: Assume that instead of forwarding a single token per round, each node is allowed to forward up to $1 < b \leq k$ tokens in each round. In

[1] To be in line with [16] and other previous work, we refer to the information pieces to be disseminated in the network as tokens.

[2] In fact, if tokens and thus also messages are restricted to a polylogarithmic number of bits, even network coding does not seem to yield more than a polylog. improvement [10, 11].

the simple token-forwarding algorithm that we described above, we can then forward a block of b tokens to every node in $n-1$ rounds and we therefore get an $O\left(\frac{nk}{b}\right)$ round upper bound. We show that every (randomized) token-forwarding algorithm needs at least $\Omega\left(n+\frac{nk}{b^2 \log n \log\log n}\right)$ rounds.

Interval Connectivity: It is natural to assume that a dynamic network cannot change arbitrarily from round to round and that some paths remain stable for a while. This is formally captured by the notion of interval connectivity as defined in [16]. A network is called T-interval connected for an integer parameter $T \geq 1$ if for any T consecutive rounds, there is a stable connected subgraph. It is shown in [16] that in a T-interval connected dynamic network, k-token dissemination can be solved in $O\left(n+\frac{nk}{T}\right)$ rounds. In this paper, we show that every (randomized) token-forwarding algorithm needs at least $\Omega\left(n+\frac{nk}{T^2 \log n}\right)$ rounds.

Vertex Connectivity: If instead of merely requiring that the network is connected in every round, we assume that the network is c-vertex connected in every round for some $c > 1$, we can also obtain a speed-up. Because in a c-vertex connected graph, every vertex cut has size at least c, if in a round all nodes that know a token t broadcast it, at least c new nodes are reached. The basic token-forwarding algorithm thus leads to an $O\left(\frac{nk}{c}\right)$ upper bound. We prove this upper bound tight up to a small factor by showing an $\Omega\left(\frac{nk}{c \log^{3/2} n}\right)$ lower bound.

δ-Partial Token Dissemination: Finally we consider the basic model, but relax the requirement on the problem by requiring that every node needs to obtain only a δ-fraction of all the k tokens for some parameter $\delta \in [0,1]$. We show that even then, at least $\Omega\left(\frac{nk\delta^3}{\log n}\right)$ rounds are needed. This also has implications for algorithms that use forward error correcting codes (FEC) to forward coded packets instead of tokens. We show that such algorithms still need at least $\Omega\left(n+k\left(\frac{n}{\log n}\right)^{1/3}\right)$ rounds until every node has received enough coded packets to decode all k tokens. Due to lack of space, the discussion of partial token dissemination, as well as some of the easier proofs for the other cases are deferred to the full version of the paper.

2 Related Work

As stated in the introduction, we use the network model introduced in [16]. That paper studies the complexity of computing basic functions such as counting the number of nodes in the network, as well as the cost the token dissemination problem that we investigate in the present paper. Previously, some basic results of the same kind were also obtained in [19] for a similar network model.

The token dissemination problem as studied here is first considered in [16] in a dynamic network setting. The paper gives a variant of the distributed $O(nk)$ token-forwarding algorithm for the case when the number of nodes n is not known. It is also shown that T-interval connectivity and always c-vertex connectivity are interesting parameters that speed up the solution by factors of $\Theta(T)$ and $\Theta(c)$, respectively. In addition, [16] gives a first $\Omega(n \log k)$ lower bound for token-forwarding algorithms in the centralized setting we study in the present paper. That lower bound is substantially

Table 1. Upper and lower bounds for token forwarding (TF) algorithms and network coding (NC) based solutions (bounds in bold are proven in this paper). All TF algorithms are distributed and deterministic while all lower bounds are for centralized randomized algorithms and a strongly adaptive adversary. The NC algorithms work either in the distributed setting against a (standard) adaptive adversary (1) or in the centralized setting against a strongly adaptive adversary (2).

	TF Alg. [16]	NC Alg. [10–12]	TF Lower Bound
always connected	nk	$O(\frac{nk}{\log n})$ (1) $O(n+k)$ (2)	$\Omega(n \log k)$ [16] $\Omega(\frac{nk}{\log n})$ [9]
T-interval conn.$^+$ T-stability*	$\frac{nk}{T}$ (+,*)	$\approx O(n+\frac{nk}{T^2})$ (1,*) $O(n+k)$ (2)	$\boldsymbol{\Omega(\frac{nk}{T^2 \log n})}$ (+)
always c-connected	$\frac{nk}{c}$	$O(\frac{nk}{c \log n})$ (1) $O(\frac{n+k}{c})$ (2)	$\boldsymbol{\Omega(\frac{nk}{c \log^{3/2} n})}$ $\boldsymbol{\Omega(\frac{nk}{c^2 \log n})}$
b-token packets	$\frac{nk}{b}$	$\approx O(\frac{nk}{b^2 \log n})$ (1) $O(n+\frac{k}{b})$ (2)	$\boldsymbol{\Omega(\frac{nk}{b^2 \log n \log\log n})}$
δ-partial token diss.	δnk	$O(\frac{\delta nk}{\log n})$ (1) $O(n+\delta k)$ (2)	$\boldsymbol{\Omega(\frac{\delta^3 nk}{\log n})}$

improved in [9], where an almost tight $\Omega(nk/\log n)$ lower bound is proven. As the lower bound from [9] is the basis of our results, we discuss it in detail in Section 4.1.

The fastest known algorithms for token dissemination in dynamic networks are based on random linear network coding. There, tokens are understood as elements (or vectors) of a finite field and in every round, every node broadcasts a random linear combination of the information it possesses. In a centralized setting, the overhead for transmitting the coefficients of the linear combination can be neglected. For this case, it is shown in [10] that in always connected dynamic networks, k tokens can be disseminated in optimal $O(n+k)$ time. If messages are large enough to store b tokens, this bound improves to again optimal $O(n+k/b)$ time. It is also possible to extend these results to always c-connected networks and to the partial token dissemination problem. Note that one possible solution for δ-partial token dissemination is to solve regular token dissemination for only δk tokens. If the overhead for coefficients is not neglected, the best known upper bounds are given in [11]. The best bounds for tokens of size $O(\log n)$, as well as the upper and lower bounds for the other scenarios are listed in Table 1. The given bound for always c-vertex connected networks is not proven in [11], it can however be obtained with similar techniques. Note also that instead of T-interval connectivity, [11] considers a somewhat stronger assumption called T-stability. In a T-stable network, the network remains fixed for intervals of length T.

Apart from token dissemination and basic aggregation tasks, other problems have been considered in the same or similar adversarial dynamic network models. In [17], the problem of coordinating actions in time is studied for always connected dynamic networks. In a recent paper, bounds on what can be achieved if the network is not always connected are discussed in [8]. For a model where nodes know their neighbors before communicating, [2] studies the time to do a random walk if the network can change adversarially. Further, the problem of gradient clock synchronization has been studied for an asynchronous variant of the model [14]. In addition, a number of papers investigate a radio network variant of essentially the dynamic network model studied here [1, 5, 15]. Another line of research looks at random dynamic networks that result from some Markov process, e.g., [3, 6, 7]. Mostly these papers analyze the time required to broadcast a single message in the network. For a more thorough discussion of related work, we refer to a recent survey [18].

3 Model and Problem Definition

In this section we introduce the dynamic network model and the token dissemination problem.

Dynamic Networks: We follow the dynamic network model of [16]: A dynamic network consists of a fixed set V of n nodes and a dynamic edge set $E : \mathbb{N} \rightarrow 2^{\{\{u,v\}|u,v \in V\}}$. Time is divided into synchronous rounds so that the network graph of round $r \geq 1$ is $G(r) = (V, E(r))$. We use the common assumption that round r starts at time $r - 1$ and it ends at time r. In each round r, every node $v \in V$ can send a message to all its neighbors in $G(r)$. Note that we assume that v has to send the same message to all neighbors, i.e., communication is by local broadcast. Also, we assume that at the beginning of a round r, when the messages are chosen, nodes are not aware of their neighborhood in $G(r)$. We typically assume that the message size is bounded by the size of a fixed number of tokens.

We say that a dynamic network $G = (V, E)$ is *always c-vertex connected* iff $G(r)$ is c-vertex connected for every round r. If a network G is always 1-vertex connected, we also say that G is *always connected*. Further, we use the definition for interval connectivity from [16]. A dynamic network is T-interval connected for an integer parameter $T \geq 1$ iff the graph $\left(V, \bigcap_{r'=r}^{r+T-1} E(r')\right)$ is connected for every $r \geq 1$. Hence, a graph is T-interval connected iff there is a stable connected subgraph for every T consecutive rounds. Note we do not assume that nodes know the stable subgraph. Also note that a dynamic graph is 1-interval connected iff it is always connected.

For our lower bound, we assume randomized algorithms and a *strongly adaptive adversary* which can decide on the network $G(r)$ of round r based on the complete history of the network up to time $r - 1$ as well as on the messages the nodes send in round r. Note that the adversary is stronger than the more typical *adaptive adversary* where the graph $G(r)$ of round r is independent of the random choices that the nodes make in round r.

The Token Dissemination Problem: We prove lower bounds on the following *token dissemination problem*. There are k tokens initially distributed among the nodes in the network (for simplicity, we assume that k is at most polynomial in n). We consider *token-forwarding algorithms* as defined in [16]. In each round, every node is allowed to broadcast $b \geq 1$ of the tokens it knows to all neighbors. Except for Section 4.2, we assume that $b = 1$. No other information about tokens can be sent, so that a node u knows exactly the tokens u kept initially and the tokens that were included in some message u received. In addition, we also consider the *δ-partial token dissemination* problem. Again, there are k tokens that are initially distributed among the nodes in the network. But here, the requirement is weaker and we only demand that in the end, every node knows a δ-fraction of the k tokens for some $\delta \in (0, 1]$.

We prove our lower bounds for centralized algorithms where a central scheduler can determine the messages sent by each node in a round r based on the initial state of all the nodes before round r. Note that lower bounds obtained for such centralized algorithms are stronger than lower bounds for distributed protocols where the message broadcast by a node u in round r only depends on the initial state of u before round r.

4 Lower Bounds

4.1 General Technique and Basic Lower Bound Proof

We start our description of the lower bound by outlining the basic techniques and by giving a slightly polished version of the lower bound proof by Dutta et al. [9]. For the discussion here, we assume that in each round, each node is allowed to broadcast a single token, i.e., $b = 1$.

In the following, we make the standard assumption that round r lasts from time $r-1$ to time r. For each node, we maintain two sets of tokens. For a time $t \geq 0$ and a node u, let $K_u(t)$ be the set of tokens known by node u at time t. In addition the adversary determines a token set $K'_u(t)$ for every node, where $K'_u(t) \subseteq K'_u(t+1)$ for all $t \geq 0$. The sets $K'_u(t)$ are constructed such that under the assumption that each node u knows the tokens $K_u(t) \cup K'_u(t)$ at time t, in round $t+1$, overall the nodes cannot learn many new tokens. Specifically, we define a potential function $\Phi(t)$ as follows:

$$\Phi(t) := \sum_{u \in V} |K_u(t) \cup K'_u(t)| . \tag{1}$$

Note that for the token dissemination problem to be completed at time T it is necessary that $\Phi(T) = nk$. Assume that at the beginning, the nodes know at most $k/2$ tokens on average, i.e., $\sum_{u \in V} |K_u(0)| \leq nk/2$. For always connected dynamic graphs, we will show that there exists a way to choose the K'-sets such that $\sum_{u \in V} |K'_u(0)| < 0.3nk$ and that for every choice of the algorithm, a simple greedy adversary can ensure that the potential grows by at most $O(\log n)$ per round. We then have $\Phi(0) \leq 0.8nk$ and since the potential needs to grow to nk, we get an $\frac{0.2nk}{O(\log n)}$ lower bound.

In each round r, for each node u, an algorithm can decide on a token to send. We denote the token sent by node u in round r by $i_u(r)$ and we call the collection of pairs $(u, i_u(r))$ for nodes $u \in V$, the token assignment of round r. Note that because a node can only broadcast a token it knows, $i_u(r) \in K_u(r-1)$ needs to hold. However, for most of the analysis, we do not make use of this fact and just consider all the k possible pairs $(u, i_u(r))$ for a node u.

If the graph $G(r)$ of round r contains the edge $\{u, v\}$, u or v learns a new token if $i_v(r) \notin K_u(r-1)$ or if $i_u(r) \notin K_v(r-1)$. Moreover, the edge $\{u, v\}$ contributes to an increase of the potential function Φ in round r if $i_v(r) \notin K_u(r-1) \cup K'_u(r-1)$ or if $i_u(r) \notin K_v(r-1) \cup K'_v(r-1)$. We call an edge $e = \{u, v\}$ *free* in round r iff the edge does not contribute to the potential difference $\Phi(r) - \Phi(r-1)$. In particular, this implies that an edge is free if

$$\big(i_u(r) \in K'_v(r-1) \wedge i_v(r) \in K'_u(r-1)\big) \vee \big(i_u(r) = i_v(r)\big). \tag{2}$$

To construct the K'-sets we use the probabilistic method. More specifically, for every token i and all nodes u, we independently put $i \in K'_u(0)$ with probability $p = 1/4$. The following lemma shows that then only a small number of non-free edges are required in every graph $G(r)$.

Lemma 1 (adapted from [9]). *If each set $K'_u(0)$ contains each token i independently with probability $p = 1/4$, for every round r and every token assignment $\{(u, i_u(r))\}$, the graph $F(r)$ induced by all free edges in round r has at most $O(\log n)$ components with probability at least $3/4$.*

Proof. Assume that the graph $F(r)$ has at least s components for some $s \geq 1$. $F(r)$ then needs to have an independent set of size s, i.e., there needs to be a set $S \subseteq V$ of size $|S| \geq s$ such that for all $u, v \in S$, the edge $\{u, v\}$ is not free in round r. Using (2) and the fact that $K'_u(0) \subseteq K'_u(t)$ for all u and $t \geq 0$, an edge $\{u, v\}$ is free in round r if $i_u(r) \in K'_v(0)$ and $i_v(r) \in K'_u(0)$ or if $i_u(r) = i_v(r)$.

To argue that s is always small we use a union bound over all $\binom{n}{s} < n^s$ ways to choose a set of s nodes and all at most k^s ways to choose the tokens to be sent out by these nodes. Note that since two nodes sending out the same token induce a free edge, all tokens sent out by nodes in S have to be distinct. Furthermore, for any pair of nodes $u, v \in S$ there is a probability of exactly p^2 for the edge $\{u, v\}$ to be free and this probability is independent for any pair u', v' with $\{u', v'\} \neq \{u, v\}$ because nodes in S send distinct tokens. The probability that all $\binom{s}{2} > s^2/4$ node pairs of S are non-free is thus exactly $(1-p^2)^{\binom{s}{2}} < e^{-p^2 s^2/4}$. If $s = 12p^{-2} \ln nk > 4p^{-2}(\ln nk + 2)$ (assuming $\ln(nk) > 1$), the union bound $(nk)^s e^{-p^2 s^2/4}$ is less than $1/4$ as desired. This shows that there is a way to choose the sets $K'_u(0)$ such that the greedy adversary always chooses a topology in which the graph $F(r)$ induced by all free edges has at most $2s \leq 24p^{-2} \ln nk = O(\log n)$ components. □

Based on Lemma 1, the lower bound from [9] now follows almost immediately.

Theorem 1. *In an always connected dynamic network with k tokens in which nodes initially know at most $k/2$ tokens on average, any centralized token-forwarding algorithm takes at least $\Omega\left(\frac{nk}{\log n}\right)$ rounds to disseminate all tokens to all nodes.*

4.2 Sending Multiple Tokens Per Round

In this section we show that it is possible to extend the lower bound to the case where nodes can send out $b > 1$ tokens in each round. Note that it is a priori not clear that this can be done as for instance the related $\Omega(n \log k)$ lower bound of [16] breaks down completely if nodes are allowed to send two instead of one tokens in each round.

In order to prove a lower bound for $b > 1$, we generalize the notion of free edges. Let us first consider a token assignment for the case $b > 1$. Instead of sending a single token $i_u(r)$, each node u now broadcasts a set $I_u(r)$ of at most b tokens in every round r. Analogously to before, we call the collection of pairs $(u, I_u(r))$ for $u \in V$, the token assignment of round r. We define the weight of an edge in round r as the amount the edge contributes to the potential function growth in round r. Hence, the weight $w(e)$ of an edge $e = \{u, v\}$ is defined as

$$w(e) := |I_v(r) \setminus (K_u(r-1) \cup K'_u(r-1))| + |I_u(r) \setminus (K_v(r-1) \cup K'_v(r-1))| . \quad (3)$$

As before, we call an edge e with weight $w(e) = 0$ free. Given the edge weights and the potential function as in Section 4.1, a simple possible strategy of the adversary works as follows. In each round, the adversary connects the nodes using an MST w.r.t. the weights $w(e)$ for all $e \in \binom{V}{2}$. The total increase of the potential function is then upper bounded by the weight of the MST.

For the MST to contain ℓ or more edges of weight at least w, there needs to be set S of $\ell + 1$ nodes such that the weight of every edge $\{u, v\}$ for $u, v \in S$ is at least w. The following lemma bounds the probability for this to happen, assuming that the K'-sets are chosen randomly such that every token i is contained in every set $K'_u(0)$ with probability $p = 1 - \varepsilon/(4eb)$ for some constant $\varepsilon > 0$.

Lemma 2. *Assume that each set $K'_u(0)$ contains each token independently with probability $1 - \varepsilon/(4eb)$. Then, for every token assignment $(u, I_u(r))$, there exists a set S of size $\ell + 1$ such that all edges connecting nodes in S have weight at least w with probability at most*

$$\exp\left((\ell+1) \cdot \left(\ln n + b \ln k + \ell + 1 - \frac{\ell w}{12} \ln\left(\frac{w}{\varepsilon}\right)\right)\right).$$

Proof. Consider an arbitrary (but fixed) set of nodes $v_0, \ldots, v_\ell$ and a set of token sets $T_0, \ldots, T_\ell$ (we assume that the token assignment contains the $\ell + 1$ pairs (v_i, T_i)). We define $\mathcal{E}_i$ to be the event that $\left|\bigcup_{j \neq i} T_j \setminus K'_{v_i}(0)\right| > \ell w/4$. Note that whenever $|K_{v_i} \cup K'_{v_i}|$ grows by more than $\ell w/4$, the event $\mathcal{E}_i$ definitely happens. In order to have $|T_j \setminus K'_{v_i}(0)| + |T_i \setminus K'_{v_j}(0)| \geq w$ for each $i \neq j$, at least $(\ell + 1)/3$ of the events $\mathcal{E}_i$ need to occur. Hence, for all edges $\{v_i, v_j\}$, $i, j \in \{0, \ldots, \ell\}$, to have weight at least w, at least $(\ell + 1)/3$ of the events $\mathcal{E}_i$ have to happen. As the event $\mathcal{E}_i$ only depends on the randomness used to determine $K'_{v_i}(0)$, events $\mathcal{E}_i$ for different i are independent. The number of events $\mathcal{E}_i$ that occur is thus dominated by a binomial random variable $\mathrm{Bin}\big(\ell+1, \max_i \mathbb{P}[\mathcal{E}_i]\big)$ variable with parameters $\ell + 1$ and $\max_i \mathbb{P}[\mathcal{E}_i]$. The probability $\mathbb{P}[\mathcal{E}_i]$ for each i can be bounded as follows:

$$\mathbb{P}[\mathcal{E}_i] \leq \binom{\ell b}{\ell w/4} \cdot \left(\frac{\varepsilon}{4eb}\right)^{\ell w/4} \leq \left(\frac{4e\ell b}{\ell w}\right)^{\ell w/4} \cdot \left(\frac{\varepsilon}{4eb}\right)^{\ell w/4} = \left(\frac{\varepsilon}{w}\right)^{\ell w/4}.$$

Let X be the number of events $\mathcal{E}_i$ that occur. We have

$$\mathbb{P}\left[X \geq \frac{\ell+1}{3}\right] \leq \binom{\ell+1}{(\ell+1)/3} \cdot \left(\frac{\varepsilon}{w}\right)^{\frac{\ell w}{4} \cdot \frac{\ell+1}{3}} \leq 2^{\ell+1} \cdot \left(\frac{\varepsilon}{w}\right)^{\frac{\ell w}{4} \cdot \frac{\ell+1}{3}}.$$

The number of possible ways to choose $\ell + 1$ nodes and assign a set of b tokens to each node is

$$\binom{n}{\ell+1} \cdot \binom{k}{b}^{\ell+1} \leq \left(nk^b\right)^{\ell+1}.$$

The claim of the lemma now follows by applying a union bound over all possible choices $v_0, \ldots, v_\ell$ and $T_0, \ldots, T_\ell$. □

Based on Lemma 2, we obtain the following theorem.

Theorem 2. *On always connected dynamic networks with k tokens in which nodes initially know at most $k/2$ tokens on average, every centralized randomized token-forwarding algorithm requires at least*

$$\Omega\left(\frac{nk}{(\log n + b\log k)b\log\log b}\right) \geq \Omega\left(\frac{nk}{b^2\log n\log\log n}\right)$$

rounds to disseminate all tokens to all nodes.

Proof. For $w_i = 2^i$, let $\ell_i + 1$ be the size of the largest set S_i, such that that edge between any two nodes $u, v \in S_i$ has weight at least w_i. Hence, in the MST, there are at most ℓ_i edges with weight between w_i and $2w_i$. The amount by which the potential function Φ increases in round r can then be upper bounded by

$$\sum_{i=0}^{\log b} 2w_i \cdot \ell_i = \sum_{i=0}^{\log b} 2^{i+1} \cdot \ell_i.$$

By Lemma 2 (and a union bound over the $\log b$ different w_i), for a sufficiently small constant $\varepsilon > 0$,

$$\ell_i = O\left(\frac{\log n + b\log k}{w_i\log w_i}\right) = O\left(\frac{\log n + b\log k}{2^i \cdot i}\right)$$

with high probability. The number of tokens learned in each round can thus be bounded by

$$\sum_{i=0}^{\log b} O\left(\frac{\log n + b\log k}{i}\right) = O\big((\log n + b\log k)\log\log b\big).$$

By a standard Chernoff bound, with high probability, the initial potential is of the order $1 - \Theta(nk/b)$. Therefore to disseminate all tokens to all nodes, the potential has to increase by $\Theta(nk/b)$ and the claim follows. □

4.3 Interval Connected Dynamic Networks

While allowing that the network can change arbitrarily from round to round is a clean and useful theoretical model, from a practical point of view it might make sense to look at dynamic graphs that are a bit more stable. In particular, some connections and paths might remain reliable over some period of time. In [16], token dissemination and the other problems considered are studied in the context of T-interval connected graphs. For T large enough, sufficiently many paths remain stable for T rounds so that it is possible to use pipelining along the stable paths to disseminate tokens significantly faster (note that this is possible even though the nodes do not know which edges are stable). In the following, we show that the lower bound described in Section 4.1 can also be extended to T-interval connected networks.

Theorem 3. *On T-interval connected dynamic networks in which nodes initially know at most $k/2$ of k tokens on average, every randomized token-forwarding algorithm requires at least*

$$\Omega\left(\frac{nk}{T(T\log k + \log n)}\right) \geq \Omega\left(\frac{nk}{T^2 \log n}\right)$$

rounds to disseminate all tokens to all nodes.

Proof. We assume that each of the sets $K'_u(0)$ independently contains each of the k tokens with probability $p = 1 - \varepsilon/T$ for a sufficiently small constant $\varepsilon > 0$. As before, we let $i_u(r)$ be the token broadcast by node u in round r and call the set of pairs $(u, i_u(r))$ the token assignment of round r. In the analysis, we will also make use of token assignments of the form $\mathcal{T} = \{(u, I_u) : u \in V\}$, where I_u is a set of tokens sent by some node u.

Given a token assignment $\mathcal{T} = \{(u, I_u)\}$, as in the previous subsection, an edge $\{u, v\}$ is free in particular if $I_u \subseteq K'_v(0) \wedge I_v \subseteq K'_u(0)$. Let $E_\mathcal{T}$ be the free edges w.r.t. a given token assignment $\mathcal{T}$. Further, we define $\mathcal{S}_\mathcal{T} = \{S_{\mathcal{T},1}, \ldots, S_{\mathcal{T},\ell}\}$ to be the partition of V induced by the components of the graph $(V, E_\mathcal{T})$.

Consider a sequence of $2T$ consecutive rounds $r_1, \ldots, r_{2T}$. For a node v_j and round r_i, $i \in [2T]$, let $I_{i,j} := \left\{i_{v_j}(r_1), \ldots, i_{v_j}(r_i)\right\}$ be the set of tokens transmitted by node v_j in rounds $r_1, \ldots, r_i$ and let $\mathcal{T}_i := \{(v_1, I_{i,1}), \ldots, (v_n, I_{i,n})\}$. As above, let $E_{\mathcal{T}_i}$ be the free edges for the token assignment $\mathcal{T}_i$ and let $\mathcal{S}_{\mathcal{T}_i}$ be the partition of V induced by the components of the graph $(V, E_{\mathcal{T}_i})$. Note that for $j > i$, $E_{\mathcal{T}_j} \subseteq E_{\mathcal{T}_i}$ and $\mathcal{S}_{\mathcal{T}_j}$ is a sub-division of $\mathcal{S}_{\mathcal{T}_i}$.

We construct edge sets $E_1, \ldots, E_{2T}$ as follows. The set E_1 contains $|\mathcal{S}_{\mathcal{T}_1}| - 1$ edges to connect the components of the graph $(V, E_{\mathcal{T}_1})$. For $i > 1$, the edge set E_i is chosen such that $E_i \subseteq E_{\mathcal{T}_{i-1}}$, $|E_i| = |\mathcal{S}_{\mathcal{T}_i}| - |\mathcal{S}_{\mathcal{T}_{i-1}}|$, and the graph $(V, E_{\mathcal{T}_i} \cup E_1 \cup \cdots \cup E_i)$ is connected. Note that such a set E_i exists by induction on i and because $\mathcal{S}_{\mathcal{T}_i}$ is a sub-division of $\mathcal{S}_{\mathcal{T}_{i-1}}$.

For convenience, we define $E_{\{r_1,\ldots,r_i\}} := E_1 \cup \cdots \cup E_i$. By the above construction, the number of edges in $E_{\{r_1,\ldots,r_i\}}$ is $|\mathcal{S}_{\mathcal{T}_i}| - 1$, where $|\mathcal{S}_{\mathcal{T}_i}|$ is the number of components of the graph $(V, E_{\mathcal{T}_i})$. Because in each round, every node transmits only one token, the number of tokens in each $I_{i,j} \in \mathcal{T}_i$ is at most $|I_{i,j}| \leq i \leq 2T$. By Lemma 2, if the constant ε is chosen small enough, the number of components of $(V, E_\mathcal{T})$ and therefore the size of $E_{\{r_1,\ldots,r_i\}}$ is upper bounded by $|\mathcal{S}_\mathcal{T}| \leq \log n + T \log k$, w.h.p.

We construct the dynamic graph as follows. For simplicity, assume that the first round of the execution is round 0. Consider some round r and let r_0 be the largest round number such that $r_0 \leq r$ and $r_0 \equiv 0 \pmod{T}$. The edge set in round i consists of the the free edges in round i, as well as of the sets $E_{i_0 - T,\ldots,i}$ and $E_{i_0,\ldots,i}$. The resulting dynamic graph is T-interval-connected. Furthermore, the number of non-free edges in each round is $O(\log n + T \log k)$. Because in each round, at most 2 tokens are learned over each non-free edge, the theorem follows. □

4.4 Vertex Connectivity

Rather than requiring more connectivity over time, we now consider the case when the network is better connected in every round. If the network is c-vertex connected for some $c > 1$, in every round, each set of nodes can potentially reach c other nodes (rather than just 1). In [16], it is shown that for the basic greedy token forwarding algorithm, one indeed gains a factor of $\Theta(c)$ if the network is c-vertex connected in every round. We first need to state two general facts about vertex connected graphs and a basic result about weighted sums of Bernoulli random variables.

Proposition 1. *If in a graph G there exists a vertex v with degree at least c such that $G - \{v\}$ is c-vertex connected then G is also c-vertex connected.*

By specializing a much more powerful result of [13], we can characterize the minimum number of edges needed to augment a graph to be c-vertex connected.

Lemma 3. *For c, any n-node graph $G = (V, E)$ with minimum degree at least $2c - 2$ can be augmented by n edges to be c-vertex connected.*

Lemma 4. *For some c let $\ell_1, \ell_2, \ldots, \ell_\tau$ be positive integers with $\ell = \sum_i \ell_i > c$. Furthermore, let $X_1, X_2, \ldots, X_\tau$ be i.i.d. Bernoulli variables with $\mathbb{P}[X_i = 1] = \mathbb{P}[X_i = 0] = 1/2$ for all i. For any integer $x > 1$ it holds that:*

$$\mathbb{P}\left[\min\left\{|L| \;:\; L \subseteq [\tau] \wedge \sum_{i \in \{j | X_j = 1\} \cup L} \ell_i \geq c\right\} > x\right] < 2^{-\Theta(\frac{x\ell}{c})}.$$

That is, the probability that x of the random variables need to be switched to one after a random assignment in order get $\sum_i X_i \ell_i \geq c$ is at most $2^{-\Theta(\frac{x\ell}{c})}$.

To prove our lower bound for always c-connected graph, we initialize the K'-sets as for always connected graphs, i.e., each token i is contained in every set $K'_u(0)$ with constant probability p (we assume $p = 1/2$ in the following). In each round, the adversary picks a c-connected graph with as few free edges as possible. Using Lemmas 1 and 3, we will show that a graph with a small number of non-free edges can be constructed as follows. First, as long as we can, we pick vertices with at least c neighbors among the remaining nodes. We then show how to extend the resulting graph to a c-connected graph.

Lemma 5. *With high probability (over the choices of the sets $K'_u(0)$), for every token assignment $(u, I_u(r))$, the largest set S for which no node $u \in S$ has at least c neighbors in S is of size $O(c \log n)$.*

Proof. Consider some round r with token assignment $\{(u, i_u(r))\}$. We need to show that for any set S of size $s = \alpha c \log n$ for a sufficiently large constant α, at least one node in S has at least c free neighbors in S (i.e., the largest degree of the graph induced by the free edges between nodes in S is at least c).

We will use a union bound over all n^s sets S and all k^s possibilities for selecting the tokens sent by these nodes. We want to show that if the constant α is chosen sufficiently large, for each of these $2^{s \log nk}$ possibilities we have a success probability of at least $1 - 2^{-2s \log nk}$.

We first partition the nodes in S according to the token sent out, i.e., S_i is the subset of nodes sending out token i. Note that if for some j we have $S_j > c$ we are done since all edges between nodes sending the same token are free. With this, let j^* be such that $\sum_{i<j^*} |S_i| \geq s/3$ and $\sum_{i>j^*} |S_i| \geq s/3$. We now claim that for every $j < j^*$, with probability at most $2^{-6|S_j|\log nk}$, there does not exist a node in S_j that has at least c free edges to nodes in $S' = \bigcup_{i>j^*} S_j$. Note that the events that a node from S_j has at least c free edges to nodes in S' are independent for different j as it only depends on which nodes u in S' have j in $K'_u(0)$ and on the $K'(0)$-sets of the nodes in S_j. The claim that we have a node with degree c in S with probability at least $1 - 2^{-2s\log nk}$ then follows from the definition of j^*.

Let us therefore consider a fixed value j. We first note that for a fixed j by standard Chernoff bounds with probability at least $1 - 2^{-\Theta(s)}$, there at least $s/3 \cdot p/2 = s/12$ nodes in S' that have token j in their initial K'-set. For α sufficiently large, this probability is at least $1 - 2^{-7c\log nk} \geq 1 - 2^{-7|S_j|\log nk}$. In the following, we assume that there are at least $s/12$ nodes u in S' for which $j \in K'_u(0)$.

Let $s_{j,i}$ for any $i > j^*$ denote the number of nodes in S_i that have token j in the initial K'-set. The number of free edges to a node u in S_j is at least $\sum_{i>j^*} X_{u,i}s_{i,j}$, where the random variable $X_{u,i}$ is 1 if node u initially has token i in $K'_u(0)$ and 0 otherwise (i.e., $X_{u,i}$ is a Bernoulli variable with parameter $1/2$). Note that since $\sum_i sj, i \geq s/12$, the expected value of the number of free edges to a node u in S_j is at least $s/24$. By a Chernoff bound, the probability that the number of free edges from a node u in S_j does not deviate by more than a constant factor with probability $1-2^{-\Theta(s/c)}$. Note that $s_{j,i} \leq c$ since $|S_j| \leq c$. For α large enough this probability is at least $1 - 2^{-7\log nk}$. Because the probability bound only depends on the choice of $K'_u(0)$, we have independence for different $u \in S_j$. Therefore, given that at least $s/12$ nodes in S' have token j, the probability that no node in S_j has at least c neighbors in S' can be upper bounded as $\left(1 - 2^{-7|S_j|\log nk}\right)$. Together with the bound on the probability that at least $s/12$ nodes in S' have token j in their $K'(0)$ set, the claim of the lemma follows. □

Lemma 5 by itself directly leads to a lower bound for token forwarding algorithms in always c-vertex connected graphs.

Corollary 1. *Suppose an always c-vertex connected dynamic network with k tokens in which nodes initially know at most a constant fraction of the tokens on average. Then, any centralized token-forwarding algorithm takes at least $\Omega\left(\frac{nk}{c^2\log n}\right)$ rounds to disseminate all tokens to all nodes.*

Proof. By Lemma 5, we know that there exists $K'(0)$-sets such that for every token assignment after adding all free edges, the size of the largest induced subgraph with maximum degree less than c is $O(c\log n)$. By Lemma 1, it suffices to make the graph induced by these $O(c\log n)$ nodes c-vertex connected to have a c-vertex connected graph on all n nodes. To achieve this, by Lemma 3, it suffices to increase all degrees to $2c-2$ and add another $O(c\log n)$ edges. Overall, the number of non-free edges we have to add for this is therefore upper bounded by $O(c^2\log n)$. Hence, the potential function increases by at most $O(c^2\log n)$ per round and since we can choose the $K'(0)$-sets so that initially the potential is at most λnk for a constant $\lambda < 1$, the bound follows. □

As shown in the following, by using a more careful analysis, we can significantly improve this lower bound for $c = \omega(\log n)$. Note that the bound given by the following theorem is at most an $O(\log^{3/2} n)$ factor away from the simple "greedy" upper bound.

Theorem 4. *Suppose an always c-vertex connected dynamic network with k tokens in which nodes initially know at most a constant fraction of the tokens on average. Then, any centralized token-forwarding algorithm takes at least $\Omega\big(\frac{nk}{c\log^{3/2} n}\big)$ rounds to disseminate all tokens to all nodes.*

Proof. We use the same construction as in Lemma 5 to obtain a set S of size $|S| = s = \alpha c \log n$ for a sufficiently large constant $\alpha > 0$ such that S needs to be augmented to a c-connected graph. Note that we want the set to be of size s and therefore we do not assume that in the induced subgraph, every node has degree less than c. We improve upon Lemma 5 by showing that it is possible to increase the potential function by adding a few more tokens to the K'-sets, so that afterwards it is sufficient to add $O(s)$ additional non-free edges to S to make the induced subgraph c-vertex connected. Hence, an important difference is that are not counting the number of edges that we need to add but the number of tokens we need to give away (i.e., add to the existing K'-sets).

We first argue that w.h.p., it is possible to raise the minimum degree of vertices in the induced subgraph of S to $2c$ without increasing the potential function by too much. Then we invoke Lemma 3 and get that at most $O(s)$ more edges are then needed to make S induce a c-connected graph as desired.

We partition the nodes in S according to the token sent out in the same way as in the proof of Lemma 5, i.e., S_i is the subset of nodes sending out token i. Let us first assume that no set S_i contains more than $s/3$ nodes. We can then divide the sets of the partition into two parts with at least $s/3$ nodes each. To argue about the sets, we rename the tokens sent out by nodes in S as $1, 2, \dots$ so that we can find a token j^* for which $\sum_{j=1}^{j^*} |S_j| \geq s/3$ and $\sum_{j>j^*} |S_j| \geq s/3$. We call the sets S_j for $j \leq j^*$ the left side of S and the sets S_j for $j > j^*$ the right side of S. If there is a set S_i with $|S_i| > s/3$, we define S_i to be the right side and all other sets S_j to be the left side of S. We will show that we can increase the potential function by at most $O(s\sqrt{\log n}) = O(c \log^{3/2} n)$ such that all the nodes on the left side have at least $2c$ neighbors on the right side. If all sets S_i are of size at most $s/3$, increasing the degrees of the nodes on the right side is then done symmetrically. If the right side consists of a single set S_i of size at least $s/3$, for α large enough we have $s/3 \geq 2c + 1$ and therefore nodes on the right side already have degree at least $2c$ by just using free edges.

We start out by adding some tokens to the sets K'_u for nodes u on the right side such that for every token $j \leq j^*$ on the left side, there are at least $s/\sqrt{\log n}$ nodes u on the right side for which $j \in K'_u$. Let us consider some fixed token $j \leq j^*$ from the left side. Because every node u on the right side has $j \in K'_u(0)$ with probability $1/2$, with probability at least $1 - 2^{-\Theta(s)}$, at least $s/\sqrt{\log n}$ nodes u on the right side have $j \in K'_u(0)$. For such a token j, we do not need to do anything. Note that the events that $j \in K'_u(0)$ are independent for different j on the left side. Therefore, for a sufficiently large constant β and a fixed collection of $\beta \log n$ tokens j sent by nodes on the left side, the probability that none of these tokens is in at least $s/\sqrt{\log n}$ sets $K'_u(0)$ for u on the right side is at most $2^{-\gamma s \log n}$ for a given constant $\gamma > 0$. As there are at most s tokens sent by nodes on the left side, the number of collections of $\beta \log n$ tokens is at most

$$\binom{s}{\beta \log n} \leq \left(\frac{es}{\beta \log n}\right)^{\beta \log n} = \left(\frac{e\alpha c}{\beta}\right)^{\beta \log n} = 2^{\Theta(\log c \log n)},$$

which is less than $2^{s \log n}$ for sufficiently large α. Hence, with probability at least $1 - 2^{-(\gamma-1)s \log n}$, for at most $\beta \log n$ tokens j on the left side there are less than $s/\sqrt{\log n}$ nodes u on the right that have $j \in K'_u(0)$. For these $O(\log n)$ tokens j, we add to j to K'_u for at most $s/\sqrt{\log n}$ nodes u on the right side, such that afterwards, for every token j sent by a node on the left side, there are at least $s/\sqrt{\log n}$ nodes u on the right for which $j \in K'_u$. Note that this increases the potential function by at most $O(s\sqrt{\log n}) = O(c \log^{3/2} n)$.

We next show that by adding another $O(c \log^{3/2} n)$ tokens to the K'-sets of the nodes on the left side, we manage to get that every node u on the left side has at least $2c$ free neighbors on the right side. For a token $j \leq j^*$ sent by some node on the left side and a token $i > j^*$ sent by some node on the right side, let $s_{i,j}$ be the number of nodes $u \in S_i$ for which $j \in K'_u$. Note that if token i is in K'_v for some $v \in S_j$, v has $s_{i,j}$ neighbors in S_i.

Using the augmentation of the K'_u-sets for nodes on the right, we have that for every $j \leq j^*$, $\sum_{i>j^*} s_{i,j} \geq s/\sqrt{\log n}$. For every $i > j^*$, with probability $1/2$, we have $i \in K'_v(0)$. In addition, we add tokens additional i to K'_v for which $i \notin K'_v(0)$ such that in the end, $\sum_{i>j^*, i \in K'_v} s_{i,j} \geq 2c$. By Lemma 4, the probability that we need to add $\geq x$ tokens is upper bounded by $2^{-\Theta\left(xs/(c\sqrt{\log n})\right)} = 2^{-\Theta\left(x\sqrt{\log n}\right)}$. As the number of tokens we need to add to K'_v is independent for different v, in total we need to add at most $O\left(\frac{s \log n}{\sqrt{\log n}}\right) = O(c \log^{3/2} n)$ tokens with probability at least $1 - 2^{-(\gamma-1)s \log n}$. Note that this is still true after a union bound over all the possible ways to distributed the $O(c \log^{3/2} n)$ tokens among the $\leq s$ nodes. Using Lemma 3, we then have to add at most $O(s) = O(c \log n)$ additional non-free edges to make the graph induced by S c-vertex connected.

There are at most $n^s = 2^{s \log n}$ ways to choose the set S and $k^s = 2^{O(s \log n)}$ ways to assign tokens to the nodes in S. Hence, if we choose γ sufficiently large, the probability that we need to increase the potential by at most $O(c \log^{3/2} n)$ for all sets S and all token assignments is positive. The theorem now follows as in the previous lower bounds (e.g., as in the proof of Theorem 1). □

Acknowledgments. We would like to thank Chinmoy Dutta, Gopal Pandurangan, Rajmohan Rajaraman, and Zhifeng Sun for helpful discussions and for sharing their work at an early stage.

References

1. Anta, A.F., Milani, A., Mosteiro, M.A., Zaks, S.: Opportunistic Information Dissemination in Mobile Ad-hoc Networks: The Profit of Global Synchrony. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 374–388. Springer, Heidelberg (2010)

2. Avin, C., Koucký, M., Lotker, Z.: How to Explore a Fast-Changing World (Cover Time of a Simple Random Walk on Evolving Graphs). In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 121–132. Springer, Heidelberg (2008)
3. Baumann, H., Crescenzi, P., Fraigniaud, P.: Parsimonious flooding in dynamic graphs. In: Proc. 28th ACM Symp. on Principles of Distributed Computing (PODC), pp. 260–269 (2009)
4. Clementi, A., Macci, C., Monti, A., Pasquale, F., Silvestri, R.: Flooding time in edge-markovian dynamic graphs. In: Proc. of 27th ACM Symp. on Principles of Distributed Computing (PODC), pp. 213–222 (2008)
5. Clementi, A., Monti, A., Pasquale, F., Silvestri, R.: Broadcasting in dynamic radio networks. Journal of Computer and System Sciences 75(4), 213–230 (2009)
6. Clementi, A., Pasquale, F., Monti, A., Silvestri, R.: Information spreading in stationary markovian evolving graphs. In: Proc. of IEEE Symp. on Parallel & Distributed Processing, IPDPS (2009)
7. Clementi, A., Silvestri, R., Trevisan, L.: Information spreading in dynamic graphs. In: Proc. 31st Symp. on Principles of Distributed Computing, PODC (2012)
8. Cornejo, A., Gilbert, S., Newport, C.: Aggregation in dynamic networks. In: Proc. 31st Symp. on Principles of Distributed Computing, PODC (2012)
9. Dutta, C., Pandurangan, G., Rajaraman, R., Sun, Z.: Information spreading in dynamic networks. CoRR, abs/1112.0384 (2011)
10. Haeupler, B.: Analyzing network coding gossip made easy. In: Proc. 43nd Symp. on Theory of Computing (STOC), pp. 293–302 (2011)
11. Haeupler, B., Karger, D.: Faster information dissemination in dynamic networks via network coding. In: Proc. 30th Symp. on Principles of Distributed Computing (PODC), pp. 381–390 (2011)
12. Haeupler, B., Médard, M.: One packet suffices - highly efficient packetized network coding with finite memory. In: 2011 IEEE International Symposium on Information Theory Proceedings (ISIT), pp. 1151–1155 (2011)
13. Jackson, B., Jordán, T.: Independence free graphs and vertex connectivity augmentation. Journal of Combinatorial Theory, Series B 94(1), 31–77 (2005)
14. Kuhn, F., Lenzen, C., Locher, T., Oshman, R.: Optimal gradient clock synchronization in dynamic networks. In: Proc. of 29th ACM Symp. on Principles of Distributed Computing (PODC), pp. 430–439 (2010)
15. Kuhn, F., Lynch, N., Newport, C., Oshman, R., Richa, A.: Broadcasting in unreliable radio networks. In: Proc. of 29th ACM Symp. on Principles of Distributed Computing (PODC), pp. 336–345 (2010)
16. Kuhn, F., Lynch, N., Oshman, R.: Distributed computation in dynamic networks. In: Proc. 42nd Symp. on Theory of Computing (STOC), pp. 557–570 (2010)
17. Kuhn, F., Moses, Y., Oshman, R.: Coordinated consensus in dynamic networks. In: Proc. 30th Symp. on Principles of Distributed Computing (PODC), pp. 1–10 (2011)
18. Kuhn, F., Oshman, R.: Dynamic networks: Models and algorithms. SIGACT News 42(1), 82–96 (2011)
19. O'Dell, R., Wattenhofer, R.: Information dissemination in highly dynamic graphs. In: Proc. of Workshop on Foundations of Mobile Computing (DIALM-POMC), pp. 104–110 (2005)

No Sublogarithmic-Time Approximation Scheme for Bipartite Vertex Cover

Mika Göös and Jukka Suomela

Helsinki Institute for Information Technology HIIT,
Department of Computer Science, University of Helsinki, Finland
{mika.goos,jukka.suomela}@cs.helsinki.fi

Abstract. König's theorem states that on bipartite graphs the size of a maximum matching equals the size of a minimum vertex cover. It is known from prior work that for every $\epsilon > 0$ there exists a *constant-time* distributed algorithm that finds a $(1+\epsilon)$-approximation of a maximum matching on 2-coloured graphs of bounded degree. In this work, we show—somewhat surprisingly—that no *sublogarithmic-time* approximation scheme exists for the dual problem: there is a constant $\delta > 0$ so that no randomised distributed algorithm with running time $o(\log n)$ can find a $(1+\delta)$-approximation of a minimum vertex cover on 2-coloured graphs of maximum degree 3. In fact, a simple application of the Linial–Saks (1993) decomposition demonstrates that this lower bound is tight.

Our lower-bound construction is simple and, to some extent, independent of previous techniques. Along the way we prove that a certain cut minimisation problem, which might be of independent interest, is hard to approximate locally on expander graphs.

1 Introduction

Many graph optimisation problems do not admit an exact solution by a fast distributed algorithm. This is true not only for most NP-hard optimisation problems, but also for problems that can be solved using sequential polynomial-time algorithms. This work is a contribution to the *distributed approximability* of such a problem: the minimum vertex cover problem on bipartite graphs—we call it 2-VC, for short.

Our focus is on *negative* results: We prove an optimal (up to constants) time lower bound $\Omega(\log n)$ for a randomised distributed algorithm to find a close-to-optimal vertex cover on bipartite 2-coloured graphs of maximum degree $\Delta = 3$. In particular, this rules out the existence of a sublogarithmic-time approximation scheme for 2-VC on sparse graphs.

Our lower bound result exhibits the following features:

– The proof is relatively simple as compared to the strength of the result; this is achieved through an application of *expander graphs* in the lower-bound construction.
– To explain the source of hardness for 2-VC we introduce a certain *distributed cut minimisation problem*, which might have applications elsewhere.

- Many previous distributed inapproximability results are based on the hardness of *local symmetry breaking.* This is not the case here: the difficulty we pinpoint for 2-VC is in the semi-global task of *gluing together* two different types of local solutions.
- Our result states that *König's theorem is non-local*—see Sect. 1.3.

1.1 The $\mathcal{LOCAL}$ Model

We work in the standard $\mathcal{LOCAL}$ model of distributed computing [10,17]. As input we are given an undirected graph $G = (V, E)$. We interpret G as defining a communication network: the nodes V host processors, and two processors can communicate directly if they are connected by an edge. All nodes run the same distributed algorithm $\mathcal{A}$. The computation of $\mathcal{A}$ on G starts out with every node $v \in V$ knowing an upper bound on $n = |V|$ and possessing a globally unique $O(\log n)$-bit identifier $\mathrm{ID}(v)$; for simplicity, we assume that $V \subseteq \{1, 2, \ldots, \mathrm{poly}(n)\}$ and $\mathrm{ID}(v) = v$. Also, we assume that the processors have access to independent (and unlimited) sources of randomness. The computation proceeds in synchronous communication rounds. In each round, all nodes first perform some local computations and then exchange (unbounded) messages with their neighbours. After some r communication rounds the nodes stop and produce local outputs. Here r is the *running time* of $\mathcal{A}$ and the output of v is denoted $\mathcal{A}(G, v)$.

The fundamental limitation of a distributed algorithm with running time r is that the output $\mathcal{A}(G, v)$ can only depend on the information available in the subgraph $G[v, r] \subseteq G$ induced on the vertices in the radius-r neighbourhood ball

$$B_G(v, r) = \{u \in V : \mathrm{dist}_G(v, u) \leq r\}.$$

Conversely, it is well known that an algorithm $\mathcal{A}$ can essentially discover the structure of $G[v, r]$ in time r. Thus, $\mathcal{A}$ can be thought of as a function mapping r-neighbourhoods $G[v, r]$ (together with the additional input labels and random bits on $B_G(v, r)$) to outputs.

While the $\mathcal{LOCAL}$ model abstracts away issues of network congestion and asynchrony, this only makes our *lower-bound* result stronger.

1.2 Our Result

Below, we concentrate on bipartite 2-coloured graphs G. That is, G is not only bipartite (which is a global property), but every node v is informed of the bipartition by an additional input label $c(v)$, where $c \colon V \to \{\mathsf{white}, \mathsf{black}\}$ is a proper 2-colouring of G.

Definition 1. *In the* 2-VC *problem we are given a 2-coloured graph* $\mathcal{G} = (G, c)$ *and the objective is to output a minimum-size vertex cover of* G.

A distributed algorithm $\mathcal{A}$ computes a vertex cover by outputting a single bit $\mathcal{A}(\mathcal{G}, v) \in \{0, 1\}$ on a node v indicating whether v is included in the solution.

This way, $\mathcal{A}$ computes the set $\mathcal{A}(G) := \{v \in V : \mathcal{A}(G, v) = 1\}$. Moreover, we say that $\mathcal{A}$ computes an *α-approximation* of 2-VC if $\mathcal{A}(G)$ is a vertex cover of G and

$$|\mathcal{A}(G)| \le \alpha \cdot \mathsf{OPT}_G,$$

where OPT_G denotes the size of a minimum vertex cover of G.

Our main result is the following.

Theorem 1. *There exists a* $\delta > 0$ *such that no randomised distributed algorithm with running time* $o(\log n)$ *can find a* $(1+\delta)$*-approximation of* 2-VC *on graphs of maximum degree* $\Delta = 3$.

A matching time upper bound follows directly from the well-known network decomposition algorithm due to Linial and Saks [11].

Theorem 2. *For every* $\epsilon > 0$ *a* $(1+\epsilon)$*-approximation of* 2-VC *can be computed with high probability in time* $O(\epsilon^{-1} \log n)$ *on graphs of maximum degree* $\Delta = O(1)$.

Proof. The subroutine *Construct_Block* in the algorithm of Linial and Saks [11] computes, in time $r = O(\epsilon^{-1} \log n)$, a set $S \subseteq V$ with the following properties. Each component in the subgraph $G[S]$ induced by S has *weak diameter* at most r, i.e., $\mathrm{dist}_G(u, v) \le r$ for each pair $u, v \in S$ belonging to the same component of $G[S]$. Moreover, they prove that, w.h.p.,

$$|S| \ge (1 - \epsilon)n.$$

Let C be a component of $G[S]$. Every node of C can discover the structure of C in time $O(r)$ by exploiting its weak diameter. Thus, every node of C can internally compute *the same* optimal solution of 2-VC on C. We can then output as a vertex cover for G the union of the optimal solutions at the components together with the vertices $V \smallsetminus S$. This results in a solution of size at most

$$\mathsf{OPT}_{G[S]} + \epsilon n \le \mathsf{OPT}_G + \epsilon n.$$

But since $\mathsf{OPT}_G \ge |E|/\Delta = \Omega(n)$ for connected G, this is a $(1 + O(\epsilon))$-approximation of 2-VC. □

1.3 König Duality

The classic theorem of König (see, e.g., Diestel [3, §2.1]) states that, on bipartite graphs, the size of a maximum matching equals the size of a minimum vertex cover. A modern perspective is to view this result through the lens of linear programming (LP) duality. The LP relaxations of these problems are *the fractional matching problem* (primal) and *the fractional vertex cover problem* (dual):

$$\begin{array}{llll} \text{maximise} & \displaystyle\sum_{e \in E} x_e & \text{minimise} & \displaystyle\sum_{v \in V} y_v \\ \text{subject to} & \displaystyle\sum_{e:\, v \in e} x_e \le 1, \ \forall v \in V & \text{subject to} & \displaystyle\sum_{v:\, v \in e} y_v \ge 1, \ \forall e \in E \\ & \mathbf{x} \ge \mathbf{0} & & \mathbf{y} \ge \mathbf{0} \end{array}$$

It is known from general LP theory (see, e.g., Papadimitriou and Steiglitz [15, §13.2]) that on bipartite graphs the above LPs do not have an integrality gap: among the optimal feasible solutions are integral vectors $\mathbf{x} \in \{0,1\}^E$ and $\mathbf{y} \in \{0,1\}^V$ that correspond to maximum matchings and minimum vertex covers.

In the context of distributed algorithms, the following is known on (bipartite) *bounded degree graphs*:

1. *Primal LP and dual LP admit local approximation schemes.* As part of their general result, Kuhn et al. [7] give a strictly local $(1+\epsilon)$-approximation scheme for the above LPs. Their algorithms run in constant time independent of the number of nodes.
2. *Integral primal admits a local approximation scheme.* Åstrand et al. [1] describe a strictly local $(1+\epsilon)$-approximation scheme for the maximum matching problem on 2-coloured graphs. Again, the running time is a constant independent of the number of nodes.
3. *Integral dual does not admit a local approximation scheme.* The present work shows—in contrast to the above positive results—that there is no local approximation scheme for 2-VC even when $\Delta = 3$.

1.4 Related Lower Bounds

There are relatively few independent methods for obtaining negative results for distributed approximation in the $\mathcal{LOCAL}$ model. We list three main sources.

Local Algorithms. Linial's [10] lower bound $\Omega(\log^* n)$ for 3-colouring a cycle together with the Ramsey technique of Naor and Stockmeyer [13] establish basic limitations on finding *exact solutions* strictly locally in constant time. These impossibility results were later extended to finding *approximate solutions* on cycle-like graphs by Lenzen and Wattenhofer [9] and Czygrinow et al. [2]. A recent work [4] generalises these techniques even further and proves that deterministic local algorithms in the $\mathcal{LOCAL}$ model are often no more powerful than algorithms running on anonymous port numbered networks. For more information on this line of research, see the survey of local algorithms [18].

Here, the inapproximability results typically exploit the inability of a local algorithm to break local symmetries. By contrast, in this work, we consider the case where the local symmetry is already broken by a 2-colouring.

KMW Bounds. Kuhn, Moscibroda and Wattenhofer [6,7,8] prove that any randomised algorithm for computing a constant-factor approximation of minimum vertex cover on general graphs requires time $\Omega(\sqrt{\log n})$ and $\Omega(\log \Delta)$. Roughly speaking, their technique consists of showing that a fast algorithm cannot tell apart two adjacent nodes v and u, even though it is globally more profitable to include v in the vertex cover and exclude u than conversely.

The lower-bound graphs of Kuhn et al. are necessarily of unbounded degree: on bounded degree graphs the set of all non-isolated nodes is a constant factor approximation of a minimum vertex cover. By contrast, our lower-bound graphs are of bounded degree and they forbid close-to-optimal approximation of 2-VC.

Sublinear-Time Centralised Algorithms. Parnas and Ron [16] discuss how a fast distributed algorithm can be used as *solution oracle* to a centralised algorithm that approximates parameters of a sparse graph G in sublinear time given a randomised query access to G. Thus, lower bounds in this model of computation also imply lower bounds for distributed algorithms. In particular, an argument of Trevisan (presented in [16]) implies that computing a $(2-\epsilon)$-approximation of a minimum vertex cover requires $\Omega(\log n)$ time on d-regular graphs, where $d = d(\epsilon)$ is sufficiently large.

We note that 2-VC is easy to approximate in this model: Nguyen and Onak [14] give a centralised constant-time algorithm to approximate the size of a maximum matching on a graph G. If we are promised that G is bipartite, a small adaptation of this algorithm approximates the size of 2-VC by König duality.

2 Deterministic Lower Bound

To best explain the basic idea of our lower bound result, we first prove Theorem 1 for a toy model that we define in Sect. 2.1; in this model, we only consider a certain class of deterministic distributed algorithms in anonymous networks. Later in Sect. 3 we will show how to implement the same proof technique in a much more general setting: randomised distributed algorithms in networks with unique identifiers.

In the present section, we find a source of hardness for 2-VC as follows. First, we argue that any approximation algorithm for the 2-VC problem also solves a certain cut minimisation problem called Recut. We then show that Recut is hard to approximate locally, which implies that 2-VC must also be hard to approximate locally.

2.1 Toy Model of Deterministic Algorithms

Throughout this section we consider deterministic algorithms $\mathcal{A}$ running in time $r = o(\log n)$ that operate on *input-labelled anonymous networks* (G, ℓ), where $G = (V, E)$ and ℓ is a labelling of V. More precisely, we impose the following additional restrictions in the $\mathcal{LOCAL}$ model:

- The nodes of G are not given random bits as input.
- The output of $\mathcal{A}$ is invariant under reassigning node identifiers. That is, if $G' = (V', E')$ is isomorphic to G via a mapping $f\colon V' \to V$, then the output of a node $v \in V$ agrees with the output of $f^{-1}(v) \in V'$:

  $$\mathcal{A}(G, \ell, v) = \mathcal{A}(G', \ell \circ f, f^{-1}(v)),$$

  where $\ell \circ f$ denotes the composition of functions ℓ and f.

Put otherwise, the only symmetry breaking information we supply $\mathcal{A}$ with is the radius-r neighbourhood topology together with the input labelling—the nodes are anonymous and do not have unique identifiers.

We will also consider graphs G that are *directed*. In this case, the directions of the edges are merely additional symmetry-breaking information; they do not restrict communication.

2.2 Recut Problem

In the following, we consider partitions of V into red and blue colour classes as determined by a labelling $\ell\colon V \to \{\mathsf{red}, \mathsf{blue}\}$. We write $\partial\ell$ for the fraction of edges crossing the red/blue cut, i.e.,

$$\partial\ell := \frac{e(\ell^{-1}(\mathsf{red}), \ell^{-1}(\mathsf{blue}))}{|E|}.$$

Definition 2. *In* the RECUT problem *we are given a labelled graph* (G, ℓ) *as input and the objective is to compute an output labelling (*a recut*)* ℓ_{out} *that minimises* $\partial\ell_{\mathsf{out}}$ *subject to the following constraints: (a) If* $\ell(V) = \{\mathsf{red}\}$*, then* $\ell_{\mathsf{out}}(V) = \{\mathsf{red}\}$*. (b) If* $\ell(V) = \{\mathsf{blue}\}$*, then* $\ell_{\mathsf{out}}(V) = \{\mathsf{blue}\}$*.*

In words, if we have an all-red input, we have to produce an all-red output, and if we have an all-blue input, we have to produce an all-blue output. Otherwise the output can be arbitrary. See Fig. 1 for an illustration.

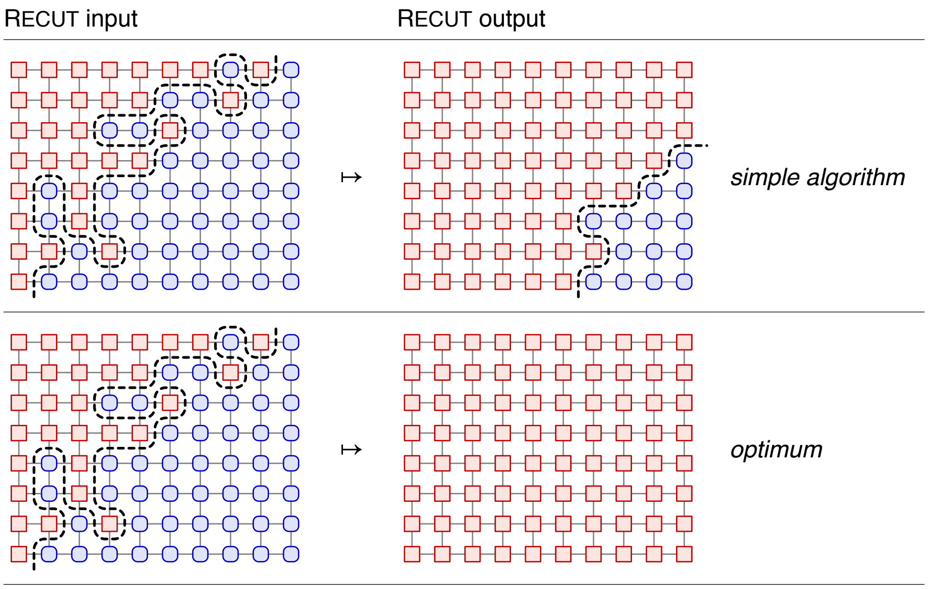


Fig. 1. The RECUT problem. In this example, we have used a simple distributed algorithm $\mathcal{A}$ to find a recut ℓ_{out} with a small boundary $\partial\ell_{\mathsf{out}}$: a node outputs red iff there is a red node within distance $r = 3$ in the input. While the solution is not optimal, in a grid graph the boundary will be relatively small. However, our lower bound shows that any fast distributed algorithm—including algorithm $\mathcal{A}$—fails to produce a small boundary in some graph.

Needless to say, the global optimum for an algorithm $\mathcal{A}$ would be to produce a constant output labelling $\ell_{\mathcal{A}}$ (either all red or all blue) having $\partial\ell_{\mathcal{A}} = 0$. However, a distributed algorithm $\mathcal{A}$ can only access the values of the input labelling ℓ in its local radius-r neighbourhood: when encountering a neighbourhood $v \in U \subseteq V$ with $\ell(U) = \{\mathsf{red}\}$, the algorithm is forced to output red at v to guarantee satisfying the global constraint (a), and when encountering a neighbourhood $v \in U \subseteq V$ with $\ell(U) = \{\mathsf{blue}\}$, the algorithm is forced to output blue at v to satisfy (b). Thus, if a connected graph G has two disjoint r-neighbourhoods $U, U' \subseteq V$ with $\ell(U) = \{\mathsf{red}\}$ and $\ell(U') = \{\mathsf{blue}\}$ the algorithm $\mathcal{A}$ cannot avoid producing at least some red/blue edge boundary. Indeed, the best we can hope $\mathcal{A}$ to achieve is a recut $\ell_{\mathcal{A}}$ of size $\partial\ell_{\mathcal{A}} \le \epsilon$ for some small constant $\epsilon > 0$.

Discussion. The RECUT problem models the following abstract high-level challenge in designing distributed algorithms: Each node in a local neighbourhood $U \subseteq V$ can, in principle, internally compute a completely *locally optimal* solution for (the subgraph induced by) U, but difficulties arise when deciding which of these proposed solution are to be used in the final distributed output. In particular, when the *type* of the produced solution changes from one (e.g., red) to another (e.g., blue) across a graph G one might have to introduce suboptimalities to the solution at the (red/blue) boundary in order to glue together the different types of local solutions.

In fact, the RECUT problem captures the first non-trivial case of this phenomenon with only *two* solution types present. One can think of the input labelling ℓ as recording the *initial preferences* of the nodes whereas the output labelling $\ell_{\mathcal{A}}$ records how an algorithm $\mathcal{A}$ decides to combine these preferences into the final unified output. In the end, our lower-bound strategy will be to argue that any $\mathcal{A}$ can be forced into producing too large an edge boundary $\partial\ell_{\mathcal{A}}$ resulting in too many suboptimalities in the produced output.

Next, we show how the above discussion is made concrete in the case of the 2-VC problem.

2.3 Reduction

We call a graph G *tree-like* if all the r-neighbourhoods in G are trees, i.e., G has girth larger than $2r + 1$. Furthermore, if G is directed, we say it is *balanced* if in-degree(v) = out-degree(v) for all vertices v. We note that a deterministic algorithm $\mathcal{A}$ produces the same output on every node of a balanced regular tree-like digraph G, because such a graph is *locally homogeneous*: all the r-neighbourhoods of G are pairwise isomorphic.

Using this terminology we give the following reduction.

Theorem 3. *Suppose $\mathcal{A}$ (with run-time r) computes a $(1 + \epsilon)$-approximation of 2-VC on graphs of maximum degree $\Delta = 3$. Then, there is an algorithm (with run-time r) that finds a recut $\ell_{\mathcal{A}}$ of size $\partial\ell_{\mathcal{A}} = O(\epsilon)$ on balanced 4-regular tree-like digraphs.*

The proof of Theorem 3 follows the usual route: We give a local reduction (i.e., one that can be computed by a local algorithm) that transforms an instance

(G, ℓ) of Recut into a white/black-coloured instance $\Pi(G, \ell)$ of 2-VC. Then we simulate $\mathcal{A}$ on the resulting instance and map the output of $\mathcal{A}$ back to a solution of the Recut instance (G, ℓ).

Let $G = (V, E)$ be a balanced 4-regular tree-like digraph and let $\ell \colon V \to \{\mathsf{red}, \mathsf{blue}\}$ be a labelling of G. The instance $\Pi(G, \ell)$ is obtained by replacing each vertex $v \in V$ by one of two local gadgets depending on the label $\ell(v)$. We first describe and analyse simple gadgets yielding instances of 2-VC with $\Delta = 4$; the gadgets yielding instances with $\Delta = 3$ are described later.

Red Gadgets. The red gadget replaces a vertex $v \in V$ by two new vertices w_v (white) and b_v (black) that share a new edge $e_v := \{w_v, b_v\}$. The incoming edges of v are reconnected to w_v, whereas the outgoing edges of v are reconnected to b_v. See Fig. 2.

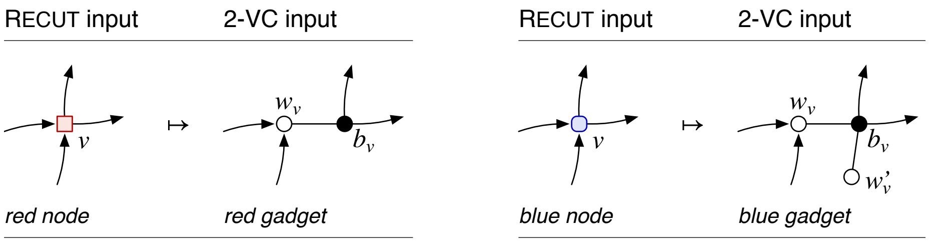


Fig. 2. Gadgets for $\Delta = 4$ (assuming an all-red input produces an all-white output)

The Case of All-Red Input. Note that the 2-VC instance $\Pi(G, \mathsf{red})$ (where we denote by red the constant labelling $v \mapsto \mathsf{red}$) contains $\{e_v : v \in V\}$ as a perfect matching. Since (G, red) is locally homogeneous, in $\Pi(G, \mathsf{red})$ the solutions output by $\mathcal{A}$ on the endpoints of e_v are isomorphic across all v. Assuming $\epsilon < 1$ it follows that the algorithm $\mathcal{A}$ must output either the set of all white nodes or the set of all black nodes on $\Pi(G, \mathsf{red})$. Our reduction branches at this point: we choose the structure of the blue gadget to counteract this white/black decision made by $\mathcal{A}$ on the red gadgets. We describe the case that $\mathcal{A}$ outputs all white nodes on $\Pi(G, \mathsf{red})$; the case of black nodes is symmetric.

Blue Gadgets. The blue gadget replacing $v \in V$ is identical to the red gadget with the exception that a third new vertex w'_v (white) is added and connected to b_v. See Fig. 2.

Similarly as above, we can argue that $\mathcal{A}$ outputs exactly the set of all black nodes on the instance $\Pi(G, \mathsf{blue})$. This completes the description of Π.

Simulation. Next, we simulate $\mathcal{A}$ on $\Pi(G, \ell)$. The output of $\mathcal{A}$ is then transformed back to a labelling $\ell_{\mathcal{A}} \colon V \to \{\mathsf{red}, \mathsf{blue}\}$ by setting

$$\ell_{\mathcal{A}}(v) = \mathsf{blue} \iff \text{the output of } \mathcal{A} \text{ contains only the black node } b_v \text{ at the gadget at } v.$$

See Fig. 3. Note that $\ell_{\mathcal{A}}$ satisfies both feasibility constraints (a) and (b) of Recut. It remains to bound the size $\partial\ell_{\mathcal{A}}$ of this recut.

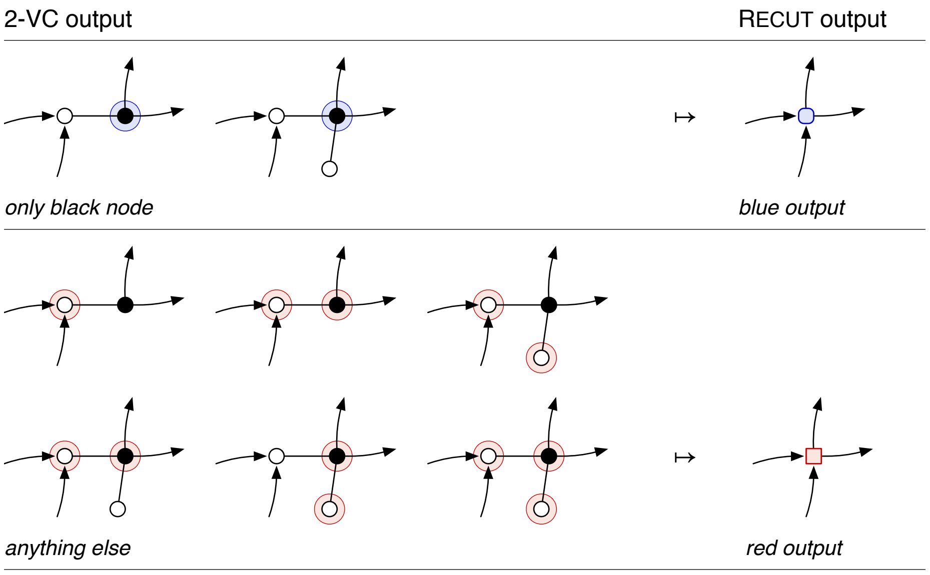


Fig. 3. Mapping the output of $\mathcal{A}$ back to a solution of the Recut problem

Recut Analysis. Call a red vertex v in $(G, \ell_{\mathcal{A}})$ *bad* if v has a blue out-neighbour u; see Fig. 4. By the definition of "$\ell_{\mathcal{A}}(u) = \text{blue}$", the vertex cover produced by algorithm $\mathcal{A}$ does not contain the white node w_u. Thus to cover the edge (b_v, w_u), the vertex cover has to contain the black node b_v. But by the definition of "$\ell_{\mathcal{A}}(v) = \text{red}$", we must have w_v or w'_v in the solution as well. Hence, at least two nodes are used to cover the gadget at v, which is suboptimal as compared to the minimum vertex cover $\{b_v : v \in V\}$, which uses only one node per gadget. This implies that we must have at most $\epsilon|V|$ bad vertices as $\mathcal{A}$ produces a $(1+\epsilon)$-approximation of 2-VC on $\Pi(G, \ell)$.

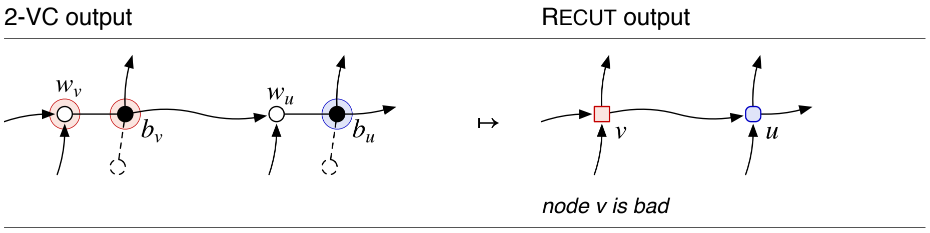


Fig. 4. A bad node: v is red and its out-neighbour u is blue

On the other hand, exactly half of the edges crossing the cut $\ell_{\mathcal{A}}$ are oriented from red to blue since G is balanced. Each bad vertex gives rise to at most two of these edges, so we have that $\partial\ell_{\mathcal{A}} \cdot |E|/2 \leq 2\epsilon|V|$ which gives $\partial\ell_{\mathcal{A}} \leq 2\epsilon$, as required. This proves Theorem 3 for $\Delta = 4$.

Gadgets for $\boldsymbol{\Delta = 3}$. The maximum degree used in the gadgets can be reduced to 3 by the following modification. The red gadget replaces a vertex $v \in V$ by a path of length 3; see Figure 5.

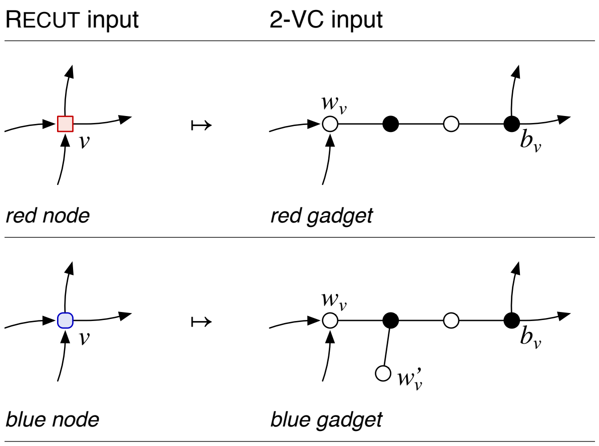


Fig. 5. Gadgets for $\Delta = 3$

Again, to achieve a 1.499-approximation of 2-VC on $\Pi(G, \mathsf{red})$ the algorithm $\mathcal{A}$ has to make a choice: either leave out the middle black vertex or the middle white vertex from the vertex cover. Supposing $\mathcal{A}$ leaves out the middle black, the blue gadget is defined to be identical to the red gadget with an additional white vertex connected to the middle black one.

After simulating $\mathcal{A}$ on an instance $\Pi(G, \ell)$ we define $\ell_{\mathcal{A}}(v) = \mathsf{blue}$ iff $\mathcal{A}$ outputs only black nodes at the gadget at v. The recut analysis will then give $\partial\ell_{\mathcal{A}} \leq 4\epsilon$.

2.4 Recut Is Hard on Expanders

Intuitively, the difficulty in computing a small red/blue cut in the Recut problem stems from the inability of an algorithm $\mathcal{A}$ to overcome the neighbourhood expansion of an input graph in $r = o(\log n)$ steps—an algorithm cannot hide the red/blue boundary as the radius-r neighbourhoods themselves might have large boundaries.

To formalise this intuition, we use as a basis for our lower-bound construction an infinite family $\mathcal{F}$ of 4-regular δ-*expander graphs*, where each $G = (V, E) \in \mathcal{F}$ satisfies the edge expansion condition

$$e(S, V \smallsetminus S) \geq \delta \cdot |S| \qquad \text{for all } S \subseteq V,\ |S| \leq n/2. \tag{1}$$

Here, $e(S, V \smallsetminus S)$ is the number of edges leaving S and $\delta > 0$ is an absolute constant independent of $n = |V|$.

To fool an algorithm $\mathcal{A}$ into producing a large recut on expanders it is enough for us to force $\mathcal{A}$ to output a *nearly balanced recut* $\ell_{\mathcal{A}}$ where both colour classes have size $n/2 \pm o(n)$. This is because if the number of, say, the red nodes is

$$|\ell_{\mathcal{A}}^{-1}(\mathsf{red})| = n/2 - o(n),$$

then the expansion property (1) implies that

$$\partial \ell_{\mathcal{A}} \geq \delta/4 - o(1).$$

That is, $\mathcal{A}$ computes a recut of size $\Omega(\delta)$.

Indeed, the following simple fooling trick makes up the very core of our argument.

Lemma 1. *Suppose $\mathcal{A}$ produces a feasible solution for the* Recut *problem in time $r = o(\log n)$. Then for each 4-regular graph G there exists an input labelling for which $\mathcal{A}$ computes a nearly balanced recut.*

Proof. Fix an arbitrary ordering $v_1, v_2, \dots, v_n$ for the vertices of G and define a sequence of labellings ℓ^0, ℓ^1, …, ℓ^n by setting $\ell^i(v_j) = \mathsf{blue}$ iff $j \leq i$. That is, in ℓ^0 all nodes are red, in ℓ^n all nodes are blue, and ℓ^i is obtained from ℓ^{i-1} by changing the colour of v_i from red to blue.

When we switch from the instance (G, ℓ^{i-1}) to (G, ℓ^i) the change of v_i's colour is only registered by nodes in the radius-r neighbourhood of v_i. This neighbourhood has size $|B_G(v_i, r)| \leq 5^r = o(n)$, and so the number of red nodes in the outputs $\ell_{\mathcal{A}}^{i-1}$ and $\ell_{\mathcal{A}}^{i}$ of $\mathcal{A}$ can only differ by $o(n)$. As, by assumption, we have that $\mathcal{A}$ computes the labelling $\ell_{\mathcal{A}}^0 = \mathsf{red}$ on (G, ℓ^0) and the labelling $\ell_{\mathcal{A}}^n = \mathsf{blue}$ on (G, ℓ^n), it follows by continuity that some labelling in our sequence must force $\mathcal{A}$ to output $n/2 - o(n)$ red nodes. □

We now have all the ingredients for the lower-bound proof: We can take $\delta = 2 - \sqrt{3}$ if we choose $\mathcal{F}$ to be the family of 4-regular Ramanujan graphs due to Morgenstern [12]. These graphs are tree-like, as they have girth $\Theta(\log n)$. They can be made into balanced digraphs since a suitable orientation can always be derived from an Euler tour. Thus, $\mathcal{F}$ consists of balanced 4-regular tree-like digraphs. Lemma 1 together with the discussion above imply that every algorithm for Recut produces a recut of size $\Omega(\delta)$ on some labelled graph in $\mathcal{F}$. Hence, the contrapositive of Theorem 3 proves Theorem 1 for our deterministic toy algorithms.

3 Randomised Lower Bound

Model. Even though our model of deterministic algorithms in Sect. 2 is an unusually weak one, we can quickly recover the standard $\mathcal{LOCAL}$ model from it by equipping the nodes with independent sources of randomness. In particular, as is well known, each node can choose an identifier uniformly at random from, e.g., the set $\{1, 2, \ldots, n^3\}$, and this results in the identifiers being globally unique with probability at least $1 - 1/n$.

For simplicity of analysis, we continue to assume

1. *Deterministic run-time:* each node runs for at most $r = o(\log n)$ steps.
2. *Las Vegas algorithm:* the algorithm always produces a feasible solution.

At a cost of only an additive $o(1)$ term in the (expected) approximation ratio we can easily modify a given algorithm that has expected running time $r' = o(\log n)$ and covers each edge with probability $1 - o(1)$ into an algorithm satisfying the above properties:

1. Choose a slowly growing function t such that $r := tr' = o(\log n)$. If a node v runs longer than r steps, we stop v's computation and output v into the vertex cover. By Markov's inequality, this modification interferes with the computation of only $o(n)$ nodes in expectation.
2. After r steps we finish by including both endpoints of each uncovered edge in the output.

Overview. When discussing randomised algorithms many of the simplifying assumptions made in Sect. 2 no longer apply. For example, a randomised algorithm need not produce the same output on every node of a locally homogeneous graph. Consequently, the homogeneous feasibility constraints in the RECUT problem do not strictly make sense for randomised algorithms.

However, we can still emulate the same proof strategy as in Sect. 2: we force the randomised algorithm to output a nearly balanced recut with high probability. Below, we describe this strategy in case of the easy-to-analyse "$\varDelta = 4$" gadgets with the understanding that the same analysis can be repeated for the "$\varDelta = 3$" gadgets with little difficulty.

3.1 Repeating Sect. 2 for Randomised Algorithms

Fix a randomised algorithm $\mathcal{A}$ with running time $r = o(\log n)$ and let $G = (V, E)$, $n = |V|$, be a large 4-regular expander.

Again, we start out with the all-red instance. We denote by W and B the number of black and white nodes output by $\mathcal{A}$ on $\varPi(G, \mathsf{red})$. As each of the edges e_v must be covered, we have that

$$W + B \geq n.$$

Hence, by linearity of expectation, at least one of $\mathbf{E}[W] \geq n/2$ or $\mathbf{E}[B] \geq n/2$ holds. We assume that $\mathbf{E}[W] \geq n/2$; the other case is symmetric.

In reaction to $\mathcal{A}$ preferring white nodes, the blue gadgets are now defined exactly as in Sect. 2. Furthermore, for any input $\ell\colon V \to \{\mathsf{red}, \mathsf{blue}\}$ we interpret the output of $\mathcal{A}$ on $\varPi(G, \ell)$ as defining an output labelling $\ell_{\mathcal{A}}$ of V, where, again,

$\ell_{\mathcal{A}}(v) = \mathsf{blue}$ iff $\mathcal{A}$ outputs only the black node at the gadget at v. This definition translates our assumption of $\mathbf{E}[W] \geq n/2$ into

$$\mathbf{E}[R(\mathsf{red})] \geq n/2, \tag{2}$$

where $R(\ell) := |\ell_{\mathcal{A}}^{-1}(\mathsf{red})|$ counts the number of gadgets (i.e., vertices of G) relabelled red by $\mathcal{A}$ on $\Pi(G, \ell)$.

If $\mathcal{A}$ relabels a blue gadget red, it must output at least two nodes at the gadget. This means that the size of the solution output by $\mathcal{A}$ on $\Pi(G, \mathsf{blue})$ is at least $n + R(\mathsf{blue})$. Thus, if $\mathcal{A}$ is to produce a 3/2-approximation on $\Pi(G, \mathsf{blue})$ in expectation, we must have that

$$\mathbf{E}[R(\mathsf{blue})] \leq n/2. \tag{3}$$

The inequalities (2) and (3) provide the necessary boundary conditions (replacing the feasibility constraints of Recut) for the argument of Lemma 1: by continuously changing the instance (G, red) into (G, blue) we may find an input labelling ℓ^* achieving

$$\mathbf{E}[R(\ell^*)] = n/2 - o(n). \tag{4}$$

It remains to argue that $\mathcal{A}$ outputs a nearly balanced recut not only "in expectation" but also with high probability.

3.2 Local Concentration Bound

Focusing on the instance $\Pi(G, \ell^*)$ we write $R = R(\ell^*)$ and

$$R = \sum_{v \in V} X_v, \tag{5}$$

where $X_v \in \{0, 1\}$ indicates whether $\mathcal{A}$ relabels the gadget at v red.

The variables X_v are not *too* dependent: the $2r$th power of G, denoted G^{2r}, where $u, v \in V$ are joined by an edge iff $B_G(v, r) \cap B_G(u, r) \neq \varnothing$, is *a dependency graph* for the variables X_v. Every independent set $I \subseteq V$ in G^{2r} corresponds to a set $\{X_v\}_{v \in I}$ of mutually independent random variables. Since the maximum degree of G^{2r} is at most $\max_v |B_G(v, 2r)| = o(n)$, this graph can always be partitioned into $\chi(G^{2r}) = o(n)$ independent sets.

Indeed, Janson [5] presents large deviation bounds for sums of type (5) by applying Chernoff–Hoeffding bounds for each colour class in a $\chi(G^{2r})$-colouring of G^{2r}. For any $\epsilon > 0$, Theorem 2.1 in Janson [5], as applied to our setting, gives

$$\mathbf{Pr}(R \geq \mathbf{E}[R] + \epsilon n) \leq \exp\left(-2 \frac{(\epsilon n)^2}{\chi(G^{2r}) \cdot n}\right) \to 0, \quad \text{as } n \to \infty, \tag{6}$$

and the same bound holds for $\mathbf{Pr}\,(R \leq \mathbf{E}[R] - \epsilon n)$. That is, R is concentrated around its expectation.

In conclusion, the combination of (4) and (6) implies that, for large n, the algorithm $\mathcal{A}$ outputs a nearly balanced recut on $\Pi(G, \ell^*)$ with high probability. By the discussion in Sect. 2, this proves Theorem 1.

Acknowledgements. Many thanks to Valentin Polishchuk for discussions, and to anonymous reviewers for their helpful comments and suggestions. This work was supported in part by the Academy of Finland, Grants 132380 and 252018.

References

1. Åstrand, M., Polishchuk, V., Rybicki, J., Suomela, J., Uitto, J.: Local algorithms in (weakly) coloured graphs (2010) (manuscript, arXiv:1002.0125 (cs.DC))
2. Czygrinow, A., Hańćkowiak, M., Wawrzyniak, W.: Fast Distributed Approximations in Planar Graphs. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 78–92. Springer, Heidelberg (2008)
3. Diestel, R.: Graph Theory, 3rd edn. Springer, Berlin (2005)
4. Göös, M., Hirvonen, J., Suomela, J.: Lower bounds for local approximation. In: Proc. 31st Symposium on Principles of Distributed Computing (PODC 2012), pp. 175–184. ACM Press, New York (2012)
5. Janson, S.: Large deviations for sums of partly dependent random variables. Random Structures & Algorithms 24(3), 234–248 (2004)
6. Kuhn, F., Moscibroda, T., Wattenhofer, R.: What cannot be computed locally! In: Proc. 23rd Symposium on Principles of Distributed Computing (PODC 2004), pp. 300–309. ACM Press, New York (2004)
7. Kuhn, F., Moscibroda, T., Wattenhofer, R.: The price of being near-sighted. In: Proc. 17th Symposium on Discrete Algorithms (SODA 2006), pp. 980–989. ACM Press, New York (2006)
8. Kuhn, F., Moscibroda, T., Wattenhofer, R.: Local computation: Lower and upper bounds (2010) (manuscript, arXiv:1011.5470 (cs.DC))
9. Lenzen, C., Wattenhofer, R.: Leveraging Linial's Locality Limit. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 394–407. Springer, Heidelberg (2008)
10. Linial, N.: Locality in distributed graph algorithms. SIAM Journal on Computing 21(1), 193–201 (1992)
11. Linial, N., Saks, M.: Low diameter graph decompositions. Combinatorica 13, 441–454 (1993)
12. Morgenstern, M.: Existence and explicit constructions of $q + 1$ regular Ramanujan graphs for every prime power q. Journal of Combinatorial Theory, Series B 62(1), 44–62 (1994)
13. Naor, M., Stockmeyer, L.: What can be computed locally? SIAM Journal on Computing 24(6), 1259–1277 (1995)
14. Nguyen, H.N., Onak, K.: Constant-time approximation algorithms via local improvements. In: Proc. 49th Symposium on Foundations of Computer Science (FOCS 2008), pp. 327–336. IEEE Computer Society Press, Los Alamitos (2008)
15. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Dover Publications, Inc., Mineola (1998)
16. Parnas, M., Ron, D.: Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms. Theoretical Computer Science 381(1-3), 183–196 (2007)
17. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia (2000)
18. Suomela, J.: Survey of local algorithms. ACM Computing Surveys (to appear), `http://www.cs.helsinki.fi/local-survey/`

"Tri, Tri Again": Finding Triangles and Small Subgraphs in a Distributed Setting (Extended Abstract)

Danny Dolev[1], Christoph Lenzen[2], and Shir Peled[1]

[1] School of Engineering and Computer Science
Hebrew University of Jerusalem, Israel
{dolev,shir.peled}@cs.huji.ac.il

[2] Department for Computer Science and Applied Mathematics
Weizmann Institute of Science, Israel
clenzen@cs.huji.ac.il

Abstract. Let $G = (V, E)$ be an n-vertex graph and M_d a d-vertex graph, for some constant d. Is M_d a subgraph of G? We consider this problem in a model where all n processes are connected to all other processes, and each message contains up to $\mathcal{O}(\log n)$ bits. A simple deterministic algorithm that requires $\mathcal{O}(n^{(d-2)/d}/\log n)$ communication rounds is presented. For the special case that M_d is a triangle, we present a probabilistic algorithm that requires an expected $\mathcal{O}(n^{1/3}/(t^{2/3}+1))$ rounds of communication, where t is the number of triangles in the graph, and $\mathcal{O}(\min\{n^{1/3}\log^{2/3} n/(t^{2/3}+1), n^{1/3}\})$ with high probability.

We also present deterministic algorithms that are specially suited for sparse graphs. In graphs of maximum degree Δ, we can test for arbitrary subgraphs of diameter D in $\mathcal{O}(\Delta^{D+1}/n)$ rounds. For triangles, we devise an algorithm featuring a round complexity of $\mathcal{O}((A^2 \log_{2+n/A^2} n)/n)$, where A denotes the arboricity of G.

1 Introduction

In distributed computing, it is common to represent a distributed system as a graph whose nodes are computational devices (or, more generally, any kind of agents) and whose edges indicate which pairs of devices can directly communicate with each other. Since its infancy, the area has been arduously studying the so-called LOCAL model (cf. [17]), where the devices try to jointly compute some combinatorial structure, such as a maximal matching or a node coloring, of this communication graph. In its most pure form, the local model is concerned with one parameter only: the locality of a problem, i.e., the number of hops up to which nodes need to learn the topology and local portions of the input in order to compute their local parts of the output—for example this could be whether or not an outgoing edge is in the maximal matching or the color of the node.

Considerable efforts have been made to understand the effect of bounding the amount of communication across each edge. In particular, the CONGEST model that demands that in each time unit, at most $\mathcal{O}(\log n)$ bits are exchanged over each edge, has been studied intensively. However, to the best of our knowledge, all known lower bounds rely on "bottlenecks" [10,12,18], i.e., small edge cuts that severely constrain the total number of bits that may be communicated between different parts of the graph. In contrast, very little is known about the possibilities and limitations in case the communication graph is a clique, i.e., the communication bounds are symmetric and *independent* of the structure of the problem we need to solve. The few existing works show that, as one can expect, such a distributed system model is very powerful: A minimum spanning tree can be found in $\mathcal{O}(\log\log n)$ time [13], with randomization nodes can send and receive up to $\mathcal{O}(n)$ messages of size $\mathcal{O}(\log n)$ in $\mathcal{O}(1)$ rounds, without any initial knowledge of which nodes hold messages for which destinations [11], and, using the latter routine, they can sort n^2 keys in $\mathcal{O}(1)$ rounds (where each node holds n keys and needs to learn their index in the sorted sequence) [16]. In general, none of these tasks can be performed fast in the local model, as the communication graph might have a large diameter.

In the current paper, we examine a question that appears to be hard even in a clique if message size is constrained to be $\mathcal{O}(\log n)$. Given that each node initially knows its neighborhood in an input graph, the goal is to decide whether this graph contains some subgraph on $d \in \mathcal{O}(1)$ vertices. In the local model, this can be trivially solved by each node learning the topology up to a constant distance;[1] in our setting, this simple strategy might result in a running time of $\Omega(n/\log n)$, as some (or all) nodes may have to learn about the entire graph and thus need to receive $\Omega(n^2)$ bits. We devise a number of algorithms that achieve much better running times. These algorithms illustrate that efficient algorithms in the contemplated model need to strive for balancing the communication load, and we show some basic strategies to do so. We will see as a corollary that it is possible for all nodes to learn about the entire graph within $\mathcal{O}(|E|/n)$ rounds and therefore locally solve any (computable) problem on the graph; this refines the immediately obvious statement that the same can be accomplished within Δ (where Δ denotes the maximum degree of G) rounds by each node sending its complete list of neighbors to all other nodes. For various settings, we achieve running times of $o(|E|/n)$ by truly distributed algorithms that do not require that (some) nodes obtain full information on the entire input.

Apart from shedding more light on the power of the considered model, the detection of small subgraphs, sometimes referred to as *graphlets* or *network motifs*, is of interest in its own right. Recently, this topic received growing attention due to the importance of recurring patterns in man-made networks as well as natural ones. Certain subgraphs were found to be associated with neurobiological networks, others with biochemical ones, and others still with human-engineered networks [15]. Detecting network motifs is an important part of understanding biological networks, for instance, as they play a key role in information processing mechanisms of biological regulation networks. Even motifs as simple as

[1] In the local model, one is satisfied with at least one node detecting a respective subgraph. Requiring that the output is known by all nodes results in the diameter being a trivial lower bound for any meaningful problem.

triangles are of interest to the biological research community as they appear in gene regulation networks, where what a graph theorist would call a directed triangle is often referred to as a Feed-Forward Loop. In recent years, the network motifs approach to studying networks lead to development of dedicated algorithms and software tools. Being of highly applicative nature, algorithms used in such context are usually researched from an experimental point of view, using naturally generated data sets [9].

Triangles and triangle-free graphs also play a central role in combinatorics. For example, it is long since known that planar triangle-free graphs are 3-colorable [8]. The implications of triangle finding and triangle-freeness motivated extensive research of algorithms, as well as lower bounds, in the centralized model. Most of the work done on these problems falls into one of two categories: subgraph listing and property testing. In subgraph listing, the aim is to list all copies of a given subgraph. The number of copies in the graph, that may be as high as $\Theta\left(n^3\right)$ for triangles, sets an obvious lower bound for the running time of such algorithms, rendering instances with many triangles harder in some sense [4]. Property testing algorithms, on the other hand, distinguish with some probability between graphs that are triangle-free and graphs that are far from being triangle-free, in the sense that a constant fraction of the edges has to be removed in order for the graph to become triangle-free [1,2]. Although soundly motivated by stability arguments, the notion of measuring the distance from triangle-freeness by the minimal number of edges that need to be removed seems less natural than counting the number of triangles in the graph. Consider for instance the case of a graph with n nodes comprised of $n-2$ triangles, all sharing the same edge. From the property testing point of view, this graph is very close to being triangle free, although it contains a linear number of triangles. Some query-based algorithms were suggested in the centralized model, where the parameter to determine is the number of triangles in the graph. The lower bounds for such algorithms assume restrictions on the type of queries[2] that cannot be justified in our model [7].

Detailed Contributions. In Section 3, we start out by giving a family of deterministic algorithms that decide whether the graph contains a d-vertex subgraph within $\mathcal{O}(n^{(d-2)/d})$ rounds. In fact, these algorithms find *all* copies of this subgraph and therefore could be used to count the exact number of occurrences. They split the task among the nodes such that each node is responsible for checking an equal number of subsets of d vertices for being the vertices of a copy of the targeted subgraph. This partition of the problem is chosen independently of the structure of the graph. Note that even the trivial algorithm that lets each node collect its D-hop neighborhood and test it for instances of the subgraph in question does not satisfy this property. Still it exhibits a structure that is simple enough to permit a deterministic implementation of running time $\mathcal{O}(\Delta^{D+1}/n)$, where Δ is the maximum degree of the graph, given in Section 4. For the special case of triangles, we present a more intricate way of checking neighborhoods that

[2] For instance, in [7] the query model requires that edges are sampled uniformly at random.

results in a running time of $\mathcal{O}(A^2/n+\log_{2+n/A^2} n) \subseteq \mathcal{O}(|E|/n+\log n)$, where the arboricity A of the graph denotes the minimal number of forests into which the edge set can be decomposed. While always $A \leq \Delta$, it is possible that $A \in \mathcal{O}(1)$, yet $\Delta \in \Theta(n)$ (e.g. in a graph that is a star). Moreover, any family of graphs excluding a fixed minor has $A \in \mathcal{O}(1)$ [5], demonstrating that the arboricity is a much less restrictive parameter than Δ. Note also that the running time bound in terms of $|E|$ is considerably weaker than the one in terms of A; it serves to demonstrate that in the worst case, the algorithm's running time essentially does not deteriorate beyond the trivial $\mathcal{O}(|E|/n)$ bound.

All our deterministic algorithms systematically check for subgraphs by either considering all possible combinations of d nodes or following the edges of the graph. If there are *many* copies of the subgraph, it can be more efficient to randomly inspect small portions of the graph. In Section 5, we present a triangle-finding algorithm that does just that, yielding that for every $\varepsilon \geq 1/n^{\mathcal{O}(1)}$ and graph containing $t \geq 1$ triangles, one will be found with probability at least $1-\varepsilon$ within $\mathcal{O}((n^{1/3}\log^{2/3}\varepsilon^{-1})/t^{2/3}+\log n)$ rounds; we show this analysis to be tight.

All our algorithms are uniform, i.e., they require no prior knowledge of parameters such as t or A. Interleaving them will result in an asymptotic running time that is bounded by the minimum of all the individual results. Due to lack of space, we only sketch most proof ideas; for details we refer to [6].

2 Model and Problem

Our model separates the computational problem from the communication model. Let $V = \{1,\ldots,n\}$ represent the nodes of a distributed system. With respect to communication, we adhere to the synchronous CONGEST model as described in [17] on the complete graph on the node set V, i.e., in each computational round, each node may send (potentially different) $\mathcal{O}(\log n)$ bits to each other node. We do not consider the amount of computation performed by each node, however, for all our algorithms it will be polynomially bounded. Instead, we measure complexity in the number of rounds until an algorithm terminates.[3] Let $G = (V,E)$ be an arbitrary graph on the same vertex set, representing the computational problem at hand. Initially, every node $i \in V$ has the list $\mathcal{N}_i := \{j \in V \mid \{i,j\} \in E\}$ of its neighbors in G, but no further knowledge of G.

The computational problem we are going to consider throughout this paper is the following: Given a graph M_d on $d \in \mathcal{O}(1)$ vertices, we wish to discover whether M_d is a subgraph of G.

3 Deterministic Algorithms for General Graphs

During our exposition, we will discuss the issues of what to communicate and how to communicate it separately. That is, given sets of $\mathcal{O}(\log n)$-sized messages at all nodes satisfying certain properties, we provide subroutines that deliver

[3] Note that ensuring termination in the same round is easy due to the full connectivity.

all messages quickly and then use these subroutines in our algorithms. We start out by giving a very efficient deterministic scheme provided that origins and destinations of all messages are initially known to *all* nodes. We then will show that this scheme can be utilized to find all triangles or other constant-sized subgraphs in sublinear time.

Full-Knowledge Message Passing. For a certain limited family of algorithms whose communication structure is basically independent of the problem graph, it is possible to exploit the full capacity of the communication system, i.e., provided that no node sends or receives more than n messages, all messages can be delivered in two rounds.

Lemma 1. *Given a bulk of messages, such that:*

1. *The source and destination of each message is known in advance to all nodes, and each source knows the contents of the messages to sent.*
2. *No node is the source of more than n messages.*
3. *No node is the destination of more than n messages.*

A routing scheme to deliver all messages within 2 rounds can be found efficiently.

To get some intuition on why Lemma 1 is true, observe that if every node initially holds a single message for every other node, the task can clearly be completed within a single round. This reduces our problem to finding a message passing scheme such that, after one round, every node will hold a single message for every other node. Now consider the bipartite multi-graph in which the vertices on the left are the nodes in their role as sources, the vertices on the right are the nodes in there role as destinations, and for each message whose source and destination are nodes i and j, respectively, there is an edge from i (on the left hand side) to j (on the right hand sight). Our desired scheme translates to finding n perfect matchings in the graph, since we could have every source send the edge used by it (i.e. the message) in the i^{th} matching to the ith node, resulting in every node holding a single message for every other node in the subsequent round. The graph described is always a disjoint union of n perfect matchings, as a corollary of Hall's Theorem. The required matchings can be found before communication commences, since all sources and destinations are known.

TriPartition - Finding Triangles Deterministically. We now present an algorithm that finds whether there are triangles in G that has a better round complexity than $\mathcal{O}(|E|/n)$ if $|E|$ is large. Apart from a possible final broadcast message informing other nodes of a discovered triangle, it exhibits the very simple communication structure required by Lemma 1.

Let $S \subseteq 2^V$ be a partition of V into equally sized subsets of cardinality $n^{2/3}$. We write $S = \{S_1, ..., S_{n^{1/3}}\}$. To each node $i \in V$ we assign a distinct triplet from S denoted $S_{i,1}, S_{i,2}, S_{i,3}$ (where repetitions are admitted). Clearly, for any subset of three nodes there is a triplet such that each node is element of one of the subsets in the triplet. Hence, for each triangle $\{t_1, t_2, t_3\}$ in G, there is some node i such that $t_1 \in S_{i,1}$, $t_2 \in S_{i,2}$, and $t_3 \in S_{i,3}$. Each node checks for

Algorithm 1. TriPartition at node i

```
1 $E_i := \emptyset$
2 for $1 \leq j < k \leq 3$ do
3     for $l \in S_{i,j}$ do
4         retrieve $\mathcal{N}_l \cap S_{i,k}$
5         for $m \in \mathcal{N}_l \cap S_{i,k}$ do $E_i := E_i \cup \{l, m\}$
6 if there exists a triangle in $G_i := (V, E_i)$ then send "triangle" to all nodes
7 if received "triangle" from some node then return true
8 else return false
```

triangles that are contained in its triplet of subsets by executing *TriPartition* (see Algorithm 1).

Theorem 1. TriPartition *correctly decides whether there exists a triangle in G and can be implemented within $\mathcal{O}(n^{1/3})$ rounds.*

Remark 1. The fact that (except for the potential final broadcast) the entire communication pattern of *TriPartition* is predefined enables to refrain from including any node identifiers into the messages. That is, instead of encoding the respective sublist of neighbors by listing their identifiers, nodes just send a $0-1$ array of bits indicating whether a node from the respective set from S is or is not a neighbor in G. The receiving node can decode the message because it is already known in advance which bit stands for which pair of nodes. We may hence improve the round complexity of *TriPartition* to $\mathcal{O}(n^{1/3}/\log n)$.

Generalization for *d*-Cliques. *TriPartition* generalizes easily to an algorithm we call *dClique0* that finds d-cliques (as well as any other subgraph on d vertices). We choose S to be a partition of V into equal size subsets of cardinality $n^{(d-1)/d}$, resulting in $S = \{S_1, ..., S_{n^{1/d}}\}$. Each node now examines the edges between all pairs of some d-sized multisubset of S (as we did for $d = 3$ in *TriPartition*). Since there are exactly $|S|^d = n$ such multisets, all possible d-cliques are examined. Every node needs to receive the list of edges for all $\binom{d}{2}$ pairs, each containing at most $(n^{(d-1)/d})^2$ edges, thus every node needs to send and receive at most $\mathcal{O}(n^{(2d-2)/d})$ messages.

Theorem 2. *dClique0 determines correctly whether there exists a d-clique (or any given d-vertex graph) in G within $\mathcal{O}(n^{(d-2)/d}/\log n)$ rounds.*

4 Finding Triangles in Sparse Graphs

In graphs that have $o(n^2)$ edges, one might hope to obtain faster algorithms. However, the algorithms from the previous section have congestion at the node level, i.e., even if there are few edges in total, some nodes may still have to send or receive lots of messages. Hence, we need different strategies for sparse graphs. In this section, we derive bounds depending on parameters that reflect the sparsity of graphs.

Algorithm 2. TriNeighbors at node i

```
1  $E_i := \emptyset$
2  for $j \in V$ s.t. $(i,j) \in E$ do
3      retrieve $\mathcal{N}_j$
4      for $k \in \mathcal{N}_j$ do $E_i := E_i \cup \{j,k\}$
5  if there exists a triangle in $G_i := (V, E_i)$ then send "triangle" to all nodes
6  if received "triangle" from some node then return true
7  else return false
```

Algorithm 3. Round-Robin-Messaging at node i

```
1  $R := \emptyset$ // collects output
2  $S := \emptyset$ // collects source nodes and #messages for $i$
3  for $j \in V$ do
4      send $m_{i,j \bmod k(i)}$ to $j$
5      if $j \in D_i$ then send "notify $k(i)$" to $j$
6  for "notify $k(j)$" received from $j$ do $S := S \cup (j, k(j))$
7  $l := 1$
8  for $(j, k(j)) \in S$ do
9      for $k \in \{1, \ldots, k(j)\}$ do
10         send "request message from $j$" to $l$
11         $l := l + 1$
12 for received "request message from $j$" do send $m_{j,i \bmod k(j)}$ to $j$
13 for received message $m$ do $R := R \cup \{m\}$
14 return $R$
```

Bounded Degree. We start with a simple value, the *maximum degree* $\Delta := \max_{i \in V} \delta_i$, where the *degree of node* i $\delta_i := |\mathcal{N}_i|$. If Δ is relatively small, then every node can simply exchange its neighbors list with all its neighbors. We refer to this as *TriNeighbors* algorithm, whose pseudo-code is given in Algorithm 2. In a graph with bounded Δ, it will be much faster than *dClique0* algorithm.

Since potentially all vertices may have degree Δ, the message complexity per node is in $\mathcal{O}(\Delta^2 + n)$. We use an elegant message-passing technique, suggested to us by Shiri Chechik [3]. Assuming that (i) no node is the source of more than n messages in total, (ii) no node is the destination of more than n messages, and (iii) every node sends the exact same messages to all of the destinations for its messages, it delivers all messages in 3 rounds. This is done by first having each node distribute its messages evenly, in a Round-Robin fashion, to all other nodes in the graph. In the second phase, messages are retrieved in a similar Round-Robin process. This divides the communication load evenly, resulting in an optimal round complexity. Assuming that for each node i we have the set of its $k(i)$ messages $M_i = \{m_{i,1}, \ldots, m_{i,k(i)}\}$, let D_i denote its recipients list. With these notations, the code of *Round-Robin Messaging* is given in Algorithm 3.

Lemma 2. *Given a bulk of messages in which:*

1. *Every node is the source of at most n messages.*
2. *Every node is the destination of at most n messages.*
3. *Every source node sends exactly the same information to all of its destination nodes and knows the content of its messages.*

Round-Robin-Messaging *delivers all messages in* 3 *rounds.*

Corollary 1. *Using* Round-Robin-Messaging*, all nodes can learn the complete structure of the graph in $|E|/n$ rounds.*

Algorithm *TriNeighbors* satisfies all the conditions of Lemma 2. We conclude that, employing *Round-Robin-Messaging*, the round complexity of *TriNeighbors* becomes $\mathcal{O}(\Delta^2/n)$. If $\Delta \in \mathcal{O}(\sqrt{n})$ then the round complexity is $\mathcal{O}(1)$, and clearly optimal. More generally, any subgraph of diameter[4] $D \in \mathcal{O}(1)$ can be detected by each node exploring its D-hop neighborhood.

Corollary 2. *We can test for subgraphs of diameter D in $\mathcal{O}(\Delta^{D+1}/n)$ rounds.*

Bounded Arboricity. The arboricity A of G is defined to be the minimum number of forests on V such that their union is G. Note that always $A \leq \Delta$, and for many graphs $A \ll \Delta$. The arboricity bounds the number of edges in any subgraph of G in terms of its nodes. We exploit this property to devise an arboricity-based algorithm for triangle finding that we call *TriArbor*.

An overview of the TriArbor algorithm. We wish to employ the same strategy used by the naive $TriNeighbors$, that is "asking neighbors for their neighbors", in a more careful manner, so as to avoid having high degree nodes send their entire neighbor list to many nodes. This is achieved by having all nodes with degree at most $4A$ send their neighbor list to their neighbors and then shut down. In the next iteration, the nodes that have a degree at most $4A$ in the graph induced by the still active nodes do the same and shut down. As $2An'$ uniformly bounds the sum of degrees of any subgraph of G containing n' nodes, in each iteration at least half of the remaining nodes is shut down. Hence, the algorithm will terminate within $\mathcal{O}(\log n)$ iterations. To control the number of messages, in each iteration we consider triangles involving at least one node of low degree (in the induced subgraph of the still active nodes). This way, any triangle will be found once any of its nodes' degrees becomes smaller than $4A$.

Obviously, no node of low degree will have to send more than $4A$ messages in this scheme. However, it may be the case that a node receives more than $4A$ messages in case it has many low-degree neighbors. To remedy that, low-degree nodes avoid sending their neighbor list to their high-degree neighbors directly, and instead send it to intermediate nodes we call *delegates*. The delegates share the load of testing their associated high-degree node's neighborhood for triangles involving a low-degree node. We present the algorithm assuming that A is known to the nodes; for details on how to remove this assumption, we refer to [6].

[4] The diameter of a graph is the maximum shortest path length over all pairs of nodes.

Algorithm 4. One iteration of TriArbor at node i

```
// compute delegates
send $\delta'_i$ to all other nodes
compute assignment of delegates to high-degree nodes and neighbor sublists
// high-degree nodes distribute neighborhood
if $\delta'_i > 4A$ then
    partition $\mathcal{N}'_i$ into $\lceil \delta'_i/4A \rceil$ lists of length at most $4A$
    send each sublist to the computed delegate
    for $j \in \mathcal{N}'_i$ do notify $j$ of delegate assigned to it // only $i$ knows order of $\mathcal{N}'_i$
// let all delegates learn about $\mathcal{N}'_j$
if $i$ is delegate of some node $j$ then
    denote by $D_j$ the set of delegates of $j$
    denote by $L_{j,i} \subset \mathcal{N}'_j$ the sublist of neighbors received from $j$
    for $k \in D_j$ do send $L_{j,i}$ to $k$
    for received sublist $L_{j,k}$ do $\mathcal{N}'_j := \mathcal{N}'_j \cup L_{j,k}$
// low-degree nodes distribute neighborhoods
if $\delta'_i \leq 4A$ then
    for $j \in \mathcal{N}'_i$ do
        if $\delta'_j \leq 4A$ then send $\mathcal{N}'_i$ to $j$ // low-degree nodes handle load alone
        else send $\mathcal{N}'_i$ to $j$'s delegate assigned to $i$
// check for triangles
for received $\mathcal{N}'_j$ (from $j$ with $\delta'_j \leq 4A$) do
    if $\mathcal{N}'_i \cap \mathcal{N}'_j \neq \emptyset$ then
        send "triangle" to all nodes // triangles with two low-degree nodes
    else if $i$ is delegate of $k$ and $\mathcal{N}'_j \cap \mathcal{N}'_k \neq \emptyset$ then
        send "triangle" to all nodes // triangle with one low-degree node
if received "triangle" then return true
else return false
```

Choosing Delegates. In each iteration, every delegate node will be assigned to a unique high-degree node, i.e., a node of degree larger than $4A$ in the subgraph induced by the nodes that are still active. In the following, we will discuss a single iteration of the algorithm. Denote by $G' := (V', E')$ some subgraph of G on n' nodes, where WLOG $V' = \{1, \ldots, n'\}$. Define δ'_i, Δ', $\mathcal{N}'_i$, etc. analogously to δ_i, Δ, $\mathcal{N}_i$, etc., but with respect to G' instead of G. We would like to assign to each node i exactly $\lceil \delta'_i/(4A) \rceil$ delegates such that each delegate is responsible for up to $4A$ of the respective high-degree node's neighbors.

This is feasible because the arboricity provides a bound of the number of nodes having at least a certain degree that is inversely proportional to this threshold, implying the following claim.

Claim. At least $n'/2$ of the nodes have degree at most $4A$ and the number of assigned delegates is bounded by n'.

Moreover, the assignment of delegates to high-degree nodes can be computed locally using a predetermined function of the degrees δ'_i. Thus, if every node communicates its degree δ_i', all nodes can determine locally the assignment of delegates to high-degree nodes in a consistent manner.

The Algorithm. Algorithm 4 shows the pseudocode of one iteration of *TriArbor*. The complete algorithm iterates until for all nodes $\delta_i' = 0$ and outputs "true" if in one of the iterations a triangle was detected and "false" otherwise.

The following claim holds because we are certain that in each iteration half of the nodes are of low degree and therefore eliminated.

Claim. *TriArbor* terminates within $\lceil \log n \rceil$ iterations.

Since triangles with a low-degree node are detected, correctness is immediate.

Lemma 3. TriArbor *correctly decides whether the graph contains a triangle.*

Round Complexity of TriArbor. We examine the round complexity of one iteration of the algorithm. Obviously, announcing degrees takes a single round only. The following claims can be veryfied by carefully applying the communication strategies we already established.

Claim. High-degree nodes' neighborhoods can be distributed in two rounds.

Claim. Exchanging neighborhood sublists between delegates can be performed in four rounds.

Claim. Low-degree nodes' neighborhoods can be communicated in $3\lceil 32A^2/n \rceil$ rounds.

Finally, announcing that a triangle is found takes one round. Recall that in each iteration at least half of the nodes have low degree and are thus eliminated. All in all, we get the following result.

Theorem 3. *Algorithm* TriArbor *is correct. It can be implemented with a running time of $\mathcal{O}(\lceil A^2/n \rceil \log n)$ rounds.*

The correctness here follows from the fact that eventually every node is eliminated, and in the respective phase it has small degree. A triangle with at least two low-degree nodes will be detected because these nodes will exchange their neighborhoods. A triangle with only one low-degree node will be detected by at least one delegate of each of its high-degree nodes.

We note that a more sophisticated implementation of the approach is uniform and slightly faster.

Corollary 3. TriArbor *can be modified to be uniform and run in $\mathcal{O}(A^2/n + \log_{2+n/A^2} n)$ rounds.*

5 Randomization

Our randomized algorithm does not exhibit an as well-structured communication pattern as the presented deterministic solutions, hence it is difficult to efficiently organize the exchange of information by means of a deterministic subroutine. Therefore, we make use of a randomized routine from [11].

Algorithm 5. TriSample at node i

```
1  $s := \sqrt{n}$ while $s < n^{1/3}$ do
2      choose uniformly random set of $s$ nodes $C_i$
3      for $j \in C_i$ do send the member list of $C_i$ to $j$
4      for received member list $C_j$ from $j$ do send $\mathcal{N}_i \cap C_j$ to $j$
5      $E_i := \emptyset$
6      for received $\mathcal{N}_j \cap C_i$ from $j$ do
7          for $k \in \mathcal{N}_j \cap C_i$ do $E_i := E_i \cup \{j, k\}$
8      if $G_i := (V, E_i)$ contains a triangle then send "triangle" to all nodes
9      if received "triangle" then return true
10     else $s := 2s$
11 run TriPartition and return its output // switch to deterministic strategy
```

Theorem 4 ([11]). *Given a bulk of messages such that:*

1. *No node is the source of more than n messages.*
2. *No node is the destination of more than n messages.*
3. *Each source knows the content of its messages.*

For any predefined constant $c > 0$, all messages can be delivered in $\mathcal{O}(1)$ rounds with high probability (w.h.p.), i.e., with probability at least $1 - 1/n^c$.

The Algorithm. When sampling randomly for triangles, we would like to use the available information as efficiently as possible. To this end, on the first iteration of Algorithm 5 all nodes sample randomly chosen induced subgraphs of a certain size and examine them for triangles. On subsequent iterations the size of the checked subgraphs is increased. Checking a subgraph of size s requires to learn about $\mathcal{O}(s^2)$ edges, while it tests for $\Theta(s^3)$ potential triangles. If $s \in \Theta(\sqrt{n})$, it thus takes a linear number of messages to collect such an induced subgraph at a node. Using the subroutine from Theorem 4, each node can sample such a graph in parallel in $\mathcal{O}(1)$ rounds. Intuitively, this means we can sample $\Theta(n^{5/2})$ subsets of three vertices in constant time. As $|\binom{V}{3}| \in \Theta(n^3)$, one therefore can expect to find a triangle quickly if at least $\Omega(\sqrt{n})$ triangles are present in G. If less triangles are in the graph, we need to sample more. In order to do this efficiently, it makes sense to increase s instead of just reiterating the routine with the same set size: The time complexity grows quadratically, whereas the number of sampled 3-vertex-subsets grows cubically. Finally, once the round complexity of an iteration hits $n^{1/3}$, we will switch to deterministic searching to guarantee termination within $\mathcal{O}(n^{1/3})$ rounds. Interestingly, the set size of $s = n^{2/3}$ corresponding to this running time ensures that even a single triangle in the graph is found with constant probability.

Round Complexity. The last iteration dominates the running time.

Lemma 4. *If* TriSample *terminates after m iterations, the round complexity is in $\mathcal{O}(2^{2m})$ with high probability.*

Our aim is to bound the number of iterations needed to detect a triangle with probability at least $1-\varepsilon$, as a function of the number of triangles in the graph. Let $T \subset \binom{V}{3}$ denote the set of triangles in G, where $|T| = t$.

On an intuitive level, the triangles are either scattered (i.e., rarely share edges) or clustered. If the triangles are scattered, then applying the inclusion-exclusion principle of the second order will give us a sufficiently strong bound on the probability of success. If the triangles are clustered, then there exists an edge that many of them share. Finding that specific edge is more likely than finding any specific triangle, and given this edge is found, the probability to find at least one of the triangles it participates in is large.

Bounding the probability of success using the inclusion-exclusion principle. We know by the inclusion-exclusion principle that

$$Pr[\text{triangle found}] \geq t \cdot Pr[\text{specific triangle found}] - \sum_{a \neq b \in T} Pr[a \text{ and } b \text{ found}].$$

For every $a \neq b \in T$ there are three cases to consider:

1. a and b are disjoint, that is $a \cap b = \emptyset$.
2. a and b share a single vertex, $|a \cap b| = 1$.
3. a and b share an edge, $|a \cap b| = 2$.

Definition 1. *For $r \in \{4,5,6\}$, $T_r \subseteq \binom{T}{2}$ is the set of pairs of distinct triangles in G that have together exactly r vertices. Denoting $t_r = |T_r|$, clearly $t_4+t_5+t_6 = \binom{t}{2} = |\binom{T}{2}|$.*

We define that $P_m = Pr[\text{triangle found in iteration } m]$ and also that $p_m = Pr[\text{node } i \text{ found triangle in iteration } m]$. For symmetry reasons the latter probability is the same for each node i.

Claim. For $0 < \varepsilon < 2$, if $p_m \geq \ln(2/\varepsilon)/n$ then $P_m \geq 1 - \varepsilon/2$.

With the above notations, from the inclusion-exclusion principle we infer that

$$p_m \geq t \cdot \left(\frac{s_m}{n - s_m + 3}\right)^3 - \sum_{k=4}^{6} t_k \cdot \left(\frac{s_m}{n - s_m}\right)^k . \tag{1}$$

Recall that we distinguish between the cases of "scattered" and "clustered" triangles. We now give these expressions a formal meaning by defining a threshold for t_4 in terms of t and a critical value $s(\varepsilon)$ of s_m. This value is defined as $s(\varepsilon) := \max\{2n^{2/3}t^{-1/3}\ln^{1/3}(2/\varepsilon), 2\sqrt{n\ln(2/\varepsilon)}\}$. The critical value stems from either of the following cases:

1. Scattered triangles - we wish to sample as many triangles as possible, and the number of triangles sampled grows cubically in s_m. The $n^{2/3}$ factor in the numerator reflects the fact that $s_m = n^{2/3}$ would imply that each triangle is sampled with constant probability.[5] Clearly having a lot of triangles in general improves the probability of success, hence the division by $t^{1/3}$.

[5] *TriPartition* selects $\mathcal{O}(n^{2/3})$ vertices per node so that all sets of 3 nodes are covered.

2. Clustered triangles - it may be the case that all triangles share a single edge, hence we must sample this edge with probability at least $1 - \varepsilon/2$. For $s_m = \sqrt{n}$ each node samples $\Theta(n)$ edges, hence each edge is sampled with constant probability.

The round complexity bound for the case of scattered triangles now follows by an application of the inclusion-exclusion principle. The probability to find a pair of triangles sharing an edge is considerably greater than finding other pairs, as only 4 vertices need to be sampled (unlike 5 or 6 if they only share a vertex or are disjoint, respectively), therefore it dominates the subtracted amount in Equality (1). More specifically, the probability that some node samples $k \in \{3, \ldots, 6\}$ fixed nodes is $\Theta((s_m/n)^k)$. Computations reveal that for the relevant values of s_m, t_5 and t_6 cannot be large enough to compensate for the factors of $(s_m/n)^2$ and $(s_m/n)^3$ by which these probabilities differ, respectively. With regard to t_4, we can give a threshold, below which the respective term can be neglected. This is done by the following Lemma that provides a probabilistic guarantee to find a triangle, assuming that there are not many pairs of triangles sharing an edge.

Lemma 5. *If $t_4 \leq tn/(2s(\varepsilon))$ and n is sufficiently large, then a triangle will be found with probability at least $1 - \varepsilon/2$ in any iteration where $s_m \geq s(\varepsilon)$.*

The strategy employed for clustered triangles is to show that due to the bound on t_4, there exists an edge shared by many triangles. An elegant proof for this fact was given by Brendan McKay [14].

Definition 2. *For each edge $e \in E$, define $\Delta_e = |\{T_i : e \subseteq T_i\}|$. In other words, Δ_e is the number of triangles that e participates in. Denote $\Delta_{\max} = \max_{e \in E} \Delta_e$.*

Lemma 6. $\Delta_{\max} \geq 2t_4/3t$.

Proof. Consider a figure consisting of two triangles sharing an edge (this is basically K_4 with one edge removed). We count the occurrences of this figure in G in two different ways:

1. Observe that t_4 counts just that.
2. Pick one of the t triangles from T, choose one of its 3 edges, denote it e. Choose one of the other $\Delta_e - 1$ triangles that share e to complete the figure. Note that this counts every figure exactly twice, since we may pick either of the two triangles in the figure to be the first one. By definition $\Delta_e - 1 \leq \Delta_{\max} - 1$, hence we count at most $3t(\Delta_{\max} - 1)/2$ occurrences.

By comparing 1. and 2., we conclude that indeed $t_4 \leq 3t(\Delta_{\max} - 1)/2$. □

Subsequently the analysis focuses on this special edge, showing that the probability to sample this edge and find a triangle containing it is sufficiently large.

Lemma 7. *If $t_4 > tn/(2 \cdot s(\varepsilon))$ then a triangle will be found with probability at least $1 - \varepsilon/2$ in any iteration where $s_m \geq s(\varepsilon)$.*

Theorem 5. *If G contains at least t triangles, for any $\varepsilon \geq 1/n^{\mathcal{O}(1)}$* TriSample *terminates within $\mathcal{O}(\min\{n^{1/3}t^{-2/3}\ln^{2/3}\varepsilon^{-1} + \ln\varepsilon^{-1}, n^{1/3}\})$ rounds with probability at least $1-\varepsilon$. It always outputs the correct result.*

Remark 2. The running time bound from Theorem 5 is asymptotically tight, that is, there are graphs for which *TriSample* runs with probability at least ε for $\Omega(n^{1/3}t^{-2/3}\ln^{2/3}\varepsilon^{-1})$ or $\Omega(\ln\varepsilon^{-1})$ rounds, respectively.

6 Conclusions

In this work, we give a number of solutions for the task of checking a distributed input graph for small subgraphs, under the assumption of an underlying fully connected communication network with bandwidth of $\mathcal{O}(\log n)$ per link. Our results show that non-trivial running time bounds can be achieved and present some communication strategies that are successful in deriving such bounds. However, we provide no insight on lower bounds for the problem, although during our work we developed intuition that strong, i.e., super-polylogarithmic, bounds might exist. Due to the very powerful model, we consider it a highly challenging task to devise such a bound. As a first step, we suggest examining the class of *oblivious* algorithms, whose communication pattern is completely predefined. Algorithm 1 is such an algorithm, and we conjecture that its running time is essentially optimal, i.e., that any deterministic oblivious algorithm deciding whether there is a triangle in the input graph must run for $\tilde{\Omega}(n^{1/3})$ rounds.

Acknowledgements. The authors would like to thank Shiri Chechik for suggesting Algorithm 3, and Brendan McKay for his proof of Lemma 6. Danny Dolev is incumbent of the Berthold Badler Chair in Computer Science. Christoph Lenzen has been supported by the Swiss National Science Foundation and the Society of Swiss Friends of the Weizmann Institute of Science. This research project was supported in part by The Israeli Centers of Research Excellence (I-CORE) program, (Center No. 4/11), by the Google Inter-university center for Electronic Markets and Auctions, and by the Kabarnit Consortium, administered by the office of the Chief Scientist of the Israeli ministry of Industry and Trade and Labor.

References

1. Alon, N.: Testing subgraphs in large graphs. Random Structures and Algorithms 21, 359–370 (2002)
2. Alon, N., Kaufman, T., Krivelevich, M., Ron, D.: Testing triangle-freeness in general graphs. SIAM Journal on Discrete Math. 22(2), 786–819 (2008)
3. Chechik, S.: Message distribution technique (2011), private communication
4. Chiba, N., Nishizeki, T.: Arboricity and subgraph listing algorithms. SIAM Journal on Computing 14, 210–223 (1985)
5. Deo, N., Litow, B.: A Structural Approach to Graph Compression. In: Proc. 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS), pp. 91–101 (1998)

6. Dolev, D., Lenzen, C., Peled, S.: "Tri, Tri again".: Finding Triangles and Small Subgraphs in a Distributed Setting. Computing Research Repository abs/1201.6652 (2012)
7. Gonen, M., Ron, D., Shavitt, Y.: Counting Stars and Other Small Subgraphs in Sublinear-Time. SIAM Journal on Discrete Mathematics 25(3), 1365–1411 (2011)
8. Grötzsch, H.: Zur Theorie der diskreten Gebilde, VII. Ein Dreifarbensatz für dreikreisfreie Netze auf der Kugel. In: Math.-Nat. Reihe., vol. 8, pp. 109–120. Wiss. Z. Martin-Luther-Univ. Halle-Wittenberg (1958/1959)
9. Kashtan, N., Itzkovitz, S., Milo, R., Alon, U.: Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. Bioinformatics 20(11), 1746–1758 (2004)
10. Kothapalli, K., Scheideler, C., Onus, M., Schindelhauer, C.: Distributed Coloring in $\tilde{\mathcal{O}}(\sqrt{\log n})$ Bit Rounds. In: IPDPS (2006)
11. Lenzen, C., Wattenhofer, R.: Tight Bounds for Parallel Randomized Load Balancing. In: Proc. 43rd Symposium on Theory of Computing (STOC), pp. 11–20 (2011)
12. Lotker, Z., Patt-Shamir, B., Peleg, D.: Distributed MST for Constant Diameter Graphs. Distributed Computing 18(6) (2006)
13. Lotker, Z., Pavlov, E., Patt-Shamir, B., Peleg, D.: MST Construction in $\mathcal{O}(loglogn)$ Communication Rounds. In: Proc. 15th Symposium on Parallel Algorithms and Architectures (SPAA), pp. 94–100 (2003)
14. McKay (mathoverflow.net/users/9025), B.: If many triangles share edges, then some edge is shared by many triangles. MathOverflow, `http://mathoverflow.net/questions/83939` (version: December 20, 2011)
15. Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, U.: Network Motifs: Simple Building Blocks of Complex Networks. Science 298(5594), 824–827 (2002), `http://dx.doi.org/10.1126/science.298.5594.824`
16. Patt-Shamir, B., Teplitsky, M.: The Round Complexity of Distributed Sorting: Extended Abstract. In: PODC, pp. 249–256 (2011)
17. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. Society for Industrial and Applied Mathematics (2000)
18. Sarma, A.D., Holzer, S., Kor, L., Korman, A., Nanongkai, D., Pandurangan, G., Peleg, D., Wattenhofer, R.: Distributed Verification and Hardness of Distributed Approximation. In: 43rd Symposium on Theory of Computing, STOC (2011)

Distributed 2-Approximation Algorithm for the Semi-matching Problem*

Andrzej Czygrinow[1], Michal Hanćkowiak[2], Edyta Szymańska[2], and Wojciech Wawrzyniak[2]

[1] School of Mathematical and Statistical Sciences, Arizona State University, Tempe, AZ,85287-1804, USA
andrzej@math.la.asu.edu

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

Abstract. In this paper we consider the problem of matching clients with servers, each of which can process a subset of clients. It is known as the *semi-matching* or *load balancing* problem in a bipartite graph $G = (V, U, E)$, where U corresponds to the clients, V to the servers, and E is the set of available connections between them. The goal is to find a set of edges $M \subseteq E$ such that every vertex in U is incident to exactly one edge in M. The *load* of a server $v \in V$ is defined as $\binom{d_M(v)+1}{2}$ where $d_M(v)$ is the degree of v in M, and the problem is to find an optimal semi-matching, i.e. a semi-matching that minimizes the sum of the loads of the servers. An optimal solution can be found sequentially in polynomial time but the distributed complexity is not well understood. Our algorithm yields $(1+\frac{1}{\alpha})$-approximation (where $\alpha = \max\left\{1, \frac{1}{2}\left(\frac{|U|}{|V|}+1\right)\right\}$) and has time complexity $O\left(\Delta^5\right)$, where Δ is the maximum degree of a vertex in V. In particular, for $\Delta = O(1)$ it gives constant approximation with constant time complexity. We also give a fast algorithm for the case when Δ is large and the degrees in V and U satisfy some additional properties. Both algorithms are deterministic.

1 Introduction

In this paper we restrict our attention to a bipartite graph $G = (V, U, E)$ with bipartition $V \cup U$ and edge set $E \subseteq V \times U$. We denote $|U| = n$ and $|V| = m$ and refer to the vertices of U as clients and to the vertices of V as servers. In what follows we assume that vertices have unique identifiers from $\{1, \ldots, n+m\}$ and know the maximum degree $\Delta = \Delta_V(G)$ of a vertex in V.

Recall that a *matching* in a bipartite graph $G = (V, U, E)$ is a set $M \subseteq E$ of disjoint edges. A matching M is called *maximal* if there is no matching M' such that M is a proper subset of M', and a matching M is called *maximum* if there is no matching M' with $|M'| > |M|$.

* The research supported by grant N N206 565740.

We are interested in a relaxation of the maximum bipartite matching problem. A *semi-matching* in a bipartite graph $G = (V, U, E)$ is a set of edges $M \subseteq E$ such that every vertex $u \in U$ is incident with exactly one edge in M. In this way a semi-matching provides an assignment of each client to a server that it is connected to. This also implies that for a semi-matching to exist each vertex in U must have degree at least one in G. For a semi-matching M and every vertex $v \in V$ we denote by $d_M(v)$ the number of edges in M incident do v, which corresponds to the number of clients that have to be processed by a server associated with v. With this setting, the total completion time (including the waiting time) of a server v for its $d_M(v)$ clients, which are served in an arbitrary sequential order is equal to $1+2+\cdots+d_M(v) = \binom{d_M(v)+1}{2}$. Therefore, we define the *cost* of a semi-matching M as

$$\text{cost}(M) = \sum_{v \in V} \binom{d_M(v)+1}{2}.$$

A semi-matching with minimum total cost is called an *optimal semi-matching*. This, in turn, corresponds to the total completion time of serving all clients by the servers.

An optimal solution to the problem can be found in polynomial time by sequential algorithms (see Sec. 1.1 for more details). In this paper we analyze the distributed complexity of the optimal semi-matching problem.

We consider a synchronous, message-passing model of computations (referred to as *LOCAL* in [11]). In this model a graph is used to represent an underlying network. The vertices of the graph correspond to computational units, and edges represent communication links. The network is synchronized and in one round a vertex can send and receive messages from all of its neighbors. In addition, in the same round, a vertex can perform some local computations. The running time of the algorithm is the number of rounds needed to solve a problem. We restrict our attention to deterministic algorithms.

1.1 Related Work

The semi-matching problem known also as the load balancing problem has been extensively studied under various names in the scheduling literature. Recently it has received renewed attention after the paper by Harvey, Ladner, Lovász, and Tamir [6], where the name *semi-matching* was introduced. In the same paper the authors proposed two sequential polynomial time algorithms. The first algorithm generalizes the Hungarian method for computing maximum bipartite matchings, while the second is based on a notion of so called *cost reducing paths*. The best running time of the latter algorithm is $O(|E|\sqrt{n+m}\log(n+m))$ and was obtained in [3]. Also, a new approach to this problem was recently proposed in [7]. The weighted version of a related problem to find a semi-matching that minimizes the maximum load among all vertices in V was considered in [3].

The problem, if solved in the distributed setting, can be used, for example, to construct a load balanced data gathering tree in sensor networks [12]. All the sequential algorithms improve an initial semi-matching to get an optimal one by using some global structures such as cost reducing paths or breadth-first search trees and cannot be applied in the *LOCAL* setting. At the same time, known distributed algorithms for the matching problem are either randomized [9] or rely on techniques that are specific to matchings [5].

As our first approach to the problem, we observed in [2] that in the distributed model of computation the optimal solution requires $\Omega(|V|)$ rounds, and proposed a greedy algorithm which yields $O(1)$-approximation of an optimal semi-matching in time $O(\Delta^2)$. In this paper the approximation ratio is reduced to two via a modification of both, the algorithm and its analysis. Moreover, we give an alternative algorithm GreedySM (see Sec. 3), which is much faster in the case when Δ is large, the degrees of vertices of V do not differ much from each other, and the degrees on U are bounded from above.

1.2 Main Result

Our main result is summarized in the following theorem. Let M^* denote an optimal semi-matching in a bipartite graph $G = (V, U, E)$.

Theorem 1. *In every bipartite graph $G = (V, U, E)$ with the maximum degree in V equal to Δ the algorithm* SemiMatch *(described in Sec. 2.2) finds a semi-matching M such that*

$$cost(M) \leq \left(1 + \frac{1}{\alpha}\right) cost(M^*), \qquad where\ \alpha = \max\left\{1, \frac{1}{2}\left(\frac{|U|}{|V|} + 1\right)\right\},$$

and the time complexity of this algorithm is $O\left(\Delta^5\right)$.

1.3 Organization

The rest of the paper is structured as follows. The next section is devoted to the main algorithm SemiMatch and its analysis for arbitrary bipartite graphs. In the last section we give a fast algorithm in the case when Δ is large and some additional conditions on the degrees of vertices of V and U are satisfied.

2 Main Algorithm

Before stating the main algorithm SemiMatch and proving Theorem 1 we need to introduce some more notation.

2.1 Notation and Non-swappable Semi-matchings

The crucial role in our proof is played by semi-matchings which are called *non-swappable*. This property is formally defined below. Observe that if for an arbitrary semi-matching M there exist two vertices $v, w \in V$ connected by a path P consisting of edges alternating between M and $E \setminus M$ and such that v is matched by M and w is not, and, moreover, $d_M(v) \geq d_M(w) + 2$ then we can lower the cost of M by switching the non-matched edges of P to M and vice-versa, i.e. by taking the symmetric difference of P and M. In such a case the path P is called *cost reducing*. It was proved in [6] that if no cost reducing path exists for a given semi-matching M then M is optimal. These paths, however, can be very long and thus impossible to detect efficiently in the distributed setting. Therefore, we restrict our attention to cost reducing paths of length two only. A semi-matching is non-swappable if there is no cost reducing path of length two.

Definition 1. *Let $G = (V, U, E)$ be a bipartite graph and M be a semi-matching. We say that M is* non-swappable *if for all $v, w \in V, u \in U$ such that $vu \in M$ and $wu \in E \setminus M$ it holds that $d_M(v) \leq d_M(w) + 1$. A path $P = vuw$ not satisfying this condition, i.e. a cost reducing path of length two, is called a* bad path*.*

Let M^* be an optimal semi-matching in $G = (V, U, E)$. Semi-matchings which are non-swappable form a good approximation of the optimal solution as it is indicated in Theorem 2 below.

Theorem 2. *For any non-swappable semi-matching M in $G = (V, U, E)$,*

$$cost(M) \leq \left(1 + \tfrac{1}{\alpha}\right) cost(M^*), \qquad where\ \alpha = \max\left\{1, \frac{1}{2}\left(\frac{|U|}{|V|} + 1\right)\right\}.$$

In order to prove Theorem 2 we need the following result.

Fact 1. *For any semi-matching M in $G = (V, U, E)$ it holds*

$$cost(M) \geq \alpha|U|, \qquad where\ \alpha = \max\left\{1, \frac{1}{2}\left(\frac{|U|}{|V|} + 1\right)\right\}.$$

Proof.

$$\text{cost}(M) = \sum_{v\in V} \binom{d_M(v)+1}{2} \geq \sum_{v\in V} d_M(v) = |U|,$$

as well as, by Cauchy-Schwartz inequality

$$\text{cost}(M) = \sum_{v\in V} \binom{d_M(v)+1}{2} \geq |V| \binom{\frac{\sum_{v\in V} d_M(v)}{|V|} + 1}{2} = \tfrac{1}{2}|U| \left(\tfrac{|U|}{|V|} + 1\right).$$

Proof (Proof of Theorem 2.). Let M be a non-swappable semi-matching and let M^* be an optimal semi-matching in G. For any $u \in U$ set v_u, v_u^* in such a way that $uv_u \in M$ and $uv_u^* \in M^*$. Define an auxiliary multi-digraph D such that for any $u \in U$ there is an arc $(v_u, v_u^*) \in D$ if $v_u \neq v_u^*$. We assume that D is connected, as otherwise we could analyze each component separately (note that cost(M) is additive). Let $d_D^+(v), d_D^-(v)$ denote the out-degree and the in-degree of vertex v respectively. Consider two subsets of V, $V^+ = \{v \in V : d_D^+(v) > d_D^-(v)\}$, $V^- = \{v \in V : d_D^-(v) > d_D^+(v)\}$. Let $d = \sum_{v\in V^+}(d^+(v) - d^-(v)) = \sum_{v\in V^-}(d^-(v) - d^+(v))$. Clearly, if $V^+ = V^- = \emptyset$ then cost$(M) =$ cost(M^*). In the other case we use the following theorem(see [1], p. 84-85) about an Eulerian cover by arc-disjoint open trails.

Theorem 3. *For every connected directed multigraph D with $d > 0$ there exist (edge disjoint) open trails $P_1, P_2, \ldots, P_d$ which cover D, i.e. $D = \bigcup_{j=1}^{d} P_j$ and $P_i \cap P_j = \emptyset$(edgewise), where the beginnings of $P_1, P_2, \ldots, P_d$ are in V^+, while the ends are in V^-.*

For every $e = (v, v^*) \in D$, let u_e be such that $v = v_{u_e}, v^* = v^*_{u_e}$. Construct a sequence of semi-matchings $M_1, \ldots, M_d$ obtained by swapping the edges of current semi-matching and the optimum M^* corresponding to the arcs of the trails. Thus, $M_1 = (M \setminus \{\{u_e v_{u_e}\} : e \in P_1\}) \cup \{\{u_e, v^*_{u_e}\} : e \in P_1\}, \ldots, M_d = (M_{d-1} \setminus \{\{u_e v_{u_e}\} : e \in P_d\}) \cup \{\{u_e, v^*_{u_e}\} : e \in P_d\} = M^*$. Let v, w be the beginning and end of P_1, respectively. Then it holds $d_M(v') = d_{M_1}(v')$ for every $v' \neq v, w$ but $d_{M_1}(v) = d_M(v) - 1$, $d_{M_1}(w) = d_M(w) + 1$ and $d_M(w) \geq d_M(v) - |P_1|$ because M is non-swappable.

So, the cost changes as follows

$$\mathrm{cost}(M) - \mathrm{cost}(M_1) = \binom{d_M(w)+1}{2} - \binom{d_M(w)+2}{2} +$$

$$+\binom{d_M(v)+1}{2} - \binom{d_M(v)}{2} = \tfrac{1}{2}\left[-2(d_M(w)+1) + 2d_M(v)\right]$$

$$= d_M(v) - d_M(w) - 1 \leq |P_1| - 1.$$

Note that M_1 is also non-swappable as no new bad paths were formed. Analogously, $\mathrm{cost}(M_{j-1}) - \mathrm{cost}(M_j) = (d_{M_{j-1}}(v) - d_{M_{j-1}}(w) - 1) \leq (|P_j| - 1)$ for every $j = 2, \ldots, d$. In consequence, $\mathrm{cost}(M) - \mathrm{cost}(M^*) \leq \sum_{j=1}^{d}(|P_j| - 1) = |D| - d \leq |D| \leq |U|$. Further, by Fact 1, we obtain $\mathrm{cost}(M) \leq \mathrm{cost}(M^*) + |D| \leq \left(1 + \frac{1}{\alpha}\right)\mathrm{cost}(M^*)$.

For arbitrary, non-swappable semi-matching the above theorem yields the following.

Corollary 1. *For any non-swappable semi-matching M in G, cost$(M) \leq 2 \cdot$ cost(M^*).*

2.2 An Approximation Algorithm and Its Analysis

We are now ready to present the algorithm SemiMatch (see the pseudocode below) returning a non-swappable semi-matching which, by Theorem 2, is a $(1 + 1/\alpha)$-approximation of the optimum. It starts by finding an arbitrary semi-matching M in $G = (V, U, E)$ (step 1) and systematically eliminating all cost reducing paths of length two with respect to M (i.e. bad paths as in Def. 1) in such a way that no new such paths are formed. This is quite a challenging task and requires a systematic approach in which we consider bad paths that end in vertices of degree $0, 1, \ldots \Delta$ in M (the loop in step 2) and remove them by the swapping operation (step 7). As a result, M is modified and possibly new bad paths are created (step 9) and are again eliminated (step 11). This process, as we show below, ends after finitely many iterations. Unfortunately, we do not know how to do it for bad paths of length greater than two, which could possibly improve the approximation ratio.

To give a formal analysis we need some more notation. Let M be a semi-matching in a bipartite graph $G = (V, U, E)$. Then we set

$$V_k := V_k(M) = \{v \in V | d_M(v) = k\}.$$

In addition, $V_{\leq k} = \{v \in V | d_M(v) \leq k\}$ and $V_{<k}, V_{\geq k}, V_{>k}$ are defined analogously. Note that as M changes during the execution of the algorithm so do the above sets. To keep track of the local values of the degrees of vertices in the algorithm we also use the sets L_k corresponding to V_k.

Recall that an M-alternating path $P = vuw$ with $vu \in M$ is bad (Def. 1) if it is cost reducing, i.e. $d_M(v) - d_M(w) \geq 2$. For every such P we set $Start(P) = \{v\}$, $End(P) = \{w\}$. For two (not necessarily disjoint) sets $A, B \subset V$ we use $Bad(A, B)$ to denote the set of all bad paths from some $x \in A$ to some $y \in B$. Observe that in this setting the condition $Bad(V, V) = \emptyset$ with respect to a current semi-matching implies that this semi-matching is non-swappable (Thm. 5). To meet this condition the algorithm eliminates bad paths by swapping, in parallel, bad paths which do not interfere with each other. Formally, we use a simple procedure denoted by $Bad_{ind}(A, B)$ to find a maximal set of paths from $Bad(A, B)$ such that for any two paths $P, P' \in Bad_{ind}(A, B)$, $Start(P) \cap Start(P') = \emptyset$ and $End(P) \cap End(P') = \emptyset$. Since M is a semi-matching, the paths constructed by $Bad_{ind}(A, B)$ are also internally disjoint. What is more, since $\Delta(V) \leq \Delta$, $Bad_{ind}(A, B)$ can be performed easily in $O(\Delta)$ rounds for any $A, B \subseteq V$. Finally, if M is a semi-matching and Z is a set of bad paths, then $Z \oplus M$ is the semi-matching obtained from M by deleting the edge $vu \in M$ and adding uv' for every path vuv' in Z. Clearly, $Z \oplus M$ is a semi-matching as the degree of every u stays one.

Algorithm 1. SemiMatch

```
 1: $\forall u \in U$ pick an arbitrary edge $e_u$ incident to $u$ and let $M = \bigcup_{u \in U} e_u$.
 2: for $k = 0$ to $\Delta - 2$ do
 3:     for $i = 0$ to $2\Delta$ do
 4:         $\forall_{v \in V}\, l(v) = d_M(v)$, $\forall_{t=0,\ldots,\Delta}\, L_t = \{v \in V | l(v) = t\}$        ▷ Layers
 5:         $X = Bad_{ind}(V_{>k+1}, V_k)$        ▷ Maximal set of disjoint paths
 6:         $S = Ends(X)$, $S^c = V \setminus S$
 7:         $M = M \oplus X$        ▷ Applying $X$ to $M$
 8:         for $j = 0$ to $2\Delta^2$ do
 9:             $Y = \bigcup_{t=1}^{k} Bad_{ind}(L_t \cap S, L_{t-1} \cap S^c)$
10:             $S = S \cup Ends(Y) \setminus Starts(Y)$, $S^c = V \setminus S$
11:             $M = M \oplus Y$        ▷ Applying $Y$ to $M$
12:         end for
13:     end for
14: end for
15: return $M$
```

We now proceed with the analysis of SemiMatch. Our goal is to prove that it terminates in $O(\Delta^5)$ steps (via Thm. 4) and returns a semi-matching that is non-swappable (via Thm. 5). At the end Theorems 4 and 5 together with Theorem 2 will yield our main result, Theorem 1.

Fix n and let $C(k,i)$ be the smallest integer C such that after C iterations of the loop 8-12 the set $Y := \bigcup_{t=1}^{k} Bad_{ind}(L_t \cap S, L_{t-1} \cap S^c)$ is empty. The following theorem provides an upper bound on the value of $C(k,i)$. It can be easily proved that $C(k,i)$ is finite. Our next result provides a specific bound in terms of Δ.

Theorem 4. *$C(k,i) \leq 2\Delta^2$.*

The next theorem guarantees that the algorithm returns a non-swappable semi-matching after termination.

Theorem 5. *After all iterations of the loop 2–14, $Bad(V,V) = \emptyset$. In particular, the semi-matching returned in Step 15 is non-swappable.*

First we show how the degree of a vertex in a semi-matching changes in the course of the algorithm. Note that in SemiMatch the label $l(v)$ of a vertex v does not change during the execution of loop 8-12 but $d_M(v)$ may change.

Fix k, i and let $d_M(v)$ denote the degree of v (in M) at the beginning of the i-th iteration (step 3) and let $d_M^{(j)}(v)$ be the degree of v at the beginning of the j-th iteration (step 8) ($j = 0, \ldots, C+1$, where $j = 2\Delta^2 + 1$ gives the degree after all iterations). For $T \in \{S, S^c\}$, we say that a vertex v *has state* T at a given time if $v \in T$ at this time.

Fact 2. *The following inequalities hold for every j.*

a) If $d_M(v) \geq k+2$, then $-1 \leq d_M^{(j)}(v) - d_M(v) \leq 0$.
b) If $d_M(v) = k+1$, then $d_M^{(j)}(v) - d_M(v) = 0$.
c) If $d_M(v) \leq k$, then $0 \leq d_M^{(j)}(v) - d_M(v) \leq 1$.

Proof. If $d_M(v) = k+1$, then v cannot be the beginning or the end of any path in X or Y and its degree does not change. If $d_M(v) \geq k+2$, then v can be only the beginning of at most one path in $X \cup Y$. Therefore, for every j, $d_M(v) \geq d_M^{(j)}(v) \geq d_M(v) - 1$. Now assume that $d_M(v) \leq k$. First, note that if $v \in S$ in step 6, then its degree increases by exactly one in step 7. Finally, in view of step 9, only paths from vertices in S to vertices in S^c are used. If $P \in Y$ and $P = vuw$, then w has state S^c, v has state S in step 9 and $v \in S^c$, $w \in S$ in step 10. Consequently, the degree of any vertex in $V_{<k+1}$ cannot decrease and can increase by at most one.

Next we show three lemmas determining the elimination of bad paths in the execution of the algorithm, which are used in the proof of Theorem 5.

Lemma 1. *For every k and i, if $Bad(V, V_{<k}) = \emptyset$ at the beginning of the i-th iteration (step 3), then $Bad(V, V_{<k}) = \emptyset$ after this iteration.*

Proof. Suppose, to the contrary, that there is a path $vuv' \in Bad(V, V_{<k})$ after the i-th iteration. First note that $v' \in \bigcup_{i \leq k} L_i$. Indeed, by Fact 2, every vertex in $V_{>k}$ has degree at least $k+1$ in all of the iterations of loop 8-12. Thus, we shall consider two cases based on $l(v)$.

- *Case 1:* $l(v) \geq k+1$. Then, in step 4, $d_M(v) \geq k+1$ and $vu \in M$ at the beginning of the iteration as v cannot be the endpoint of any path in X or Y. Thus vuv' is in $Bad(V, V_{<k})$ at the beginning of the i-th iteration contradicting the assumption.
- *Case 2:* $l(v) \leq k$. First suppose that $vu \notin M$ at the beginning of the iteration. Then, at the same time there must be another path wuv with $wu \in M$ for some w that satisfies $l(w) > l(v)$. Since $l(v) > l(v')$, $wuv' \in Bad(V, V_{<k})$ at the beginning of the iteration. Thus, we may assume that $vu \in M$ at the beginning of the iteration (step 9). Since there are no bad paths in step 5, we have $l(v) = l(v') + 1$ and after all iterations $v \in S, v' \in S^c$, that is vuv' could be added to Y, contradicting the fact that after C iterations Y is empty.

Lemma 2. *Let $Bad(V, V_{\leq k-1}) = \emptyset$ at the beginning of the k-th iteration (step 2) for some k. Then for every $i \in \{0, \ldots, 2\Delta\}$ and every $v \in V_k$ after steps 4-12 of the i-th iteration at least one of the following conditions is satisfied.*

a) $Bad(V, v) = \emptyset$.
b) $|Bad(V, v)|$ *decreases by at least one in the i-th iteration.*
c) $\max\{d_M(w) | w \in Starts(Bad(V, v))\}$ *decreases by at least one in the i-th iteration.*

Proof. Fix k. First assume that $v \notin L_k$ at the beginning of the i-th iteration (step 4) for some i. Fact 2 implies that $l(v) = k-1$. If at the end of the i-th iteration there exists a path $wuv \in Bad(V, v)$, then $l(w) > k+1$ and so $wu \in M$ in step 4 as w cannot be the endpoint of any bad path in the i-th iteration. Thus $wuv \in Bad(V, v)$ at the beginning of the k-th iteration contradicting the assumption. Thus, $Bad(V, v) = \emptyset$ in the i-th iteration.

Now suppose that $v \in L_k$ at the beginning of the i-th iteration (step 4). First, we prove that no new paths are added to $Bad(V, v)$ in the i-th iteration. Suppose, to the contrary, that wuv is added to $Bad(V, v)$. Since the degree of v has not changed in this iteration, either wu is added to M in this iteration or the degree of w (in M) increases by one. However, $w \in V_{>k}$ and no new edges incident to w are added to M excluding both possibilities, so the path wuv cannot be added in this iteration. Assume $Bad(V, v) \neq \emptyset$. If $v \in Ends(X)$ after step 6, then at least one path from $Bad(V, v)$ is deleted in step 7. If $v \notin Ends(X)$, by maximality of X, for every $x \in Starts(Bad(V, v))$, $x \in Starts(X)$. Since $l(x) > k+1$, $x \notin Ends(Y)$ for any Y and so $d_M(x)$ decreases by at least one in this iteration.

Lemma 3. *Let $Bad(V, V_{\leq k-1}) = \emptyset$ at the beginning of the k-th iteration for some k. Then $Bad(V, V_{\leq k}) = \emptyset$ after the k-th iteration.*

Proof. Assume $Bad(V, V_{\leq k-1}) = \emptyset$ at the beginning of the k-th iteration. In view of Lemma 1, $Bad(V, V_{\leq k-1}) = \emptyset$ after the k-th iteration. Let $v \in V_k$

after the k-th iteration. By Lemma 2, in each of the iterations of the loop 3-12 at least one of the conditions is satisfied and once $Bad(V,v)=\emptyset$, we cannot add new bad paths to $Bad(V,v)$. Since $d_G(w) \le \Delta$ for every $w \in V$, $0 \le |Bad(V,v)| + \max\{d_M(w) | w \in Starts(Bad(V,v))\} \le 2\Delta$ and so, by Lemma 2, after 2Δ iterations $Bad(V,v)=\emptyset$.

Proof (Proof of Theorem 5). First note that after the k-th iteration, $Bad(V,V_{\le k})=\emptyset$. Indeed, by induction on k, if $k=0$, then $Bad(V,V_{-1})=\emptyset$ and so, by Lemma 3, $Bad(V,V_{\le 0})=\emptyset$. For the inductive step, assume that $Bad(V,V_{\le k-1})=\emptyset$ and thus, by Lemma 3, $Bad(V,V_{\le k})=\emptyset$. Since $\Delta(V)\le\Delta$ and there are no bad paths ending in vertices of degree at least $\Delta-1$, after all iterations of the algorithm we have $Bad(V,V)=Bad(V,V_{\le\Delta-2})=\emptyset$.

Next we establish some facts and lemmas necessary to prove Theorem 4. To prove that $C(k,i)\le 2\Delta^2$ we will use the following auxiliary directed multigraph. Fix k,i and assume that $Bad(V,V_{<k})=\emptyset$ at the beginning of the i-th iteration of 3-13. Let $H=H_{i,k}=(V_H,E_H)$ be defined as follows, $V_H=L_{\le k}$ and for $v,w\in V_H$ there is an arc from v to w (with label u) if the path vuw was is in Y in one of the iterations 8-12. Note that if $(v,w)\in E_H$, then $l(v)>l(w)$ and $vu\in M$, $uw\notin M$ prior to applying vuw to M. We first state the following simple fact about H.

Fact 3. *The multigraph H is acyclic and the longest directed path in H has length at most $\Delta-2$. The maximum out-degree of H and the maximum in-degree of H are at most Δ.*

Proof. If (v,w) is an arc, then $l(v)>l(w)$ and so there are no cycles and the longest path in H has length at most $\Delta-2$ as $k\le\Delta-2$. There is at most one arc in H from v to w which is labeled with u and since the maximum degree of vertices in V is Δ, the max out-degree in H is at most Δ. The same applies to the in-degree.

Recall that for $T\in\{S,S^c\}$, we say that a vertex v has state T at a given time if $v\in T$ at this time. For a directed path P in H, let $q(P)$ be the sum of all changes of states of all vertices in P. In view of Fact 3 we have

$$0\le q(P)\le 2\Delta^2. \tag{1}$$

Now consider the set S in step 6. In step 7 and step 11, if $vuw\in X$ (or Y), then w acquires the S-state that was previously on v and we say that S *moves from v to w*. In addition, we say that we *apply* the path vuw. We need the following important lemma.

Lemma 4. *Fix k,i and assume that $Bad(V,V_{<k})=\emptyset$ at the beginning of the i-th iteration (steps 3-13). If there exists a bad path vuv' such that $v\in S$ and $v'\in S^c$ for some iteration j_0 (of the loop 8-12) then vuv' is a bad path in every iteration $j<j_0$ of the loop 8-12.*

Proof. Fix j_0 and suppose that vuv' is a bad path such that $v \in S$ and $v' \in S^c$ in the iteration j_0. Then, since vuv' is a bad path in one of the iterations, Fact 2 implies $d_M(v) \geq d_M(v') + 2$ in the iteration j_0. If $uv \notin M$ for some $j < j_0$, then there is w with $l(w) > l(v)$ such that $wu \in M$ for all $j < j_0$ and wuv' is a path in $Bad(V, V_{<k})$ at the beginning of the i-th iteration contradicting the assumption of the lemma.

Returning to the main line of reasoning towards the proof of Theorem 4 fix k and i. Further, for a vertex $v \in S$ in step 6, let $P = v_1 \dots v_s$ be the path of successive moves of the state S that was originally on $v_1 := v$. Thus, in step 6, v_1 has S and in steps 8-12, S will move by applying bad paths $v_i u_i v_{i+1}$ until it reaches v_s which is its final destination. To prove that in $O(\Delta^2)$ steps S will reach v_s we extend P to $P' = v_1, \dots v_s, \dots, v_{s+l}$ using the following operation. Let j_p be the largest index j such that in the j-th iteration state S on v_{s+p} is moved to a vertex w and let v_{s+p+1} be this vertex w. Continue extending P' if possible. Note that P' is a directed path in the multigraph H introduced earlier. The following lemma holds for P'.

Lemma 5. *For every iteration of the loop* $8 - 12$ *either at least one vertex on* P' *changes its state or state* S *is on its final destination* v_s.

Proof. Consider an iteration of the loop 8-12 and suppose that at the beginning of this iteration the state S that originated at v_1 is on v_i for some $i \in \{1, \dots, s-1\}$. We prove that there exists a $j > i$ such that $v_j \in P'$ has state S^c. (Note that i, j have now nothing to do with the indices of the loops in the algorithm.) First observe that if for every $j \in \{s+1, \dots, s+p\}$, the state of v_j is S, then these are final states of these vertices. By maximality of P', the vertex v_{s+p} has its final state. Now suppose, $v_{s+i}, \dots, v_{s+p}$ have their final states (all S). If $v_{s+i-1} \in S$, then the state S cannot move from v_{s+i-1} to v_{s+i} and since, by definition of P', the vertex v_{s+i} was the last recipient of the S-state from v_{s+i-1}. So v_{s+i-1} will not change its state. Now, if all $v_{s+1}, \dots, v_{s+p}$ are in state S, then v_s is in S^c as by definition of P', v_{s+1} was the last recipient of an S-state from v_s.

Further, since $i < j$, v_i has state S, and v_j has state S^c, there is an index $i \leq l < j$ such that $v_l \in S$, $v_{l+1} \in S^c$. Thus, by Lemma 4, the path $v_l u_l v_{l+1}$ is a bad path in this iteration. By maximality of Y in step 9, either v_l or v_{l+1} is the endpoint of a path from Y.

Proof (Proof of Theorem 4). For a fixed k, i and a vertex $v \in S$ in step 6, let P' be defined as before. By (1), $q(P') \leq 2\Delta^2$ and, in view of Lemma 5, in $2\Delta^2$ iterations the state originated at v will reach its final destination. Since every bad path is obtained from some $v \in S$, after $2\Delta^2$ iterations there will be no bad paths in Y, that is $Y = \emptyset$.

3 Semi-matchings via the Minimum Sum Set Cover

In this section we present an alternative approach to the problem of computing a semi-matching in a distributed setting. We give an algorithm for approximating the optimal semi-matching with a slightly modified definition of the cost (see the proof of Theorem 6 for details) in a graph $G = (V, U, E)$ with degrees on U bounded from above and with an additional assumption on degrees of vertices in V. The method relies on a reduction to the *Minimum Sum Set Cover* (MSSC) problem that we shall define next. Given a hypergraph $H = (V_H, E_H)$, a solution to the MSSC problem is a bijection $\phi : V_H \to \{1, \ldots, |V_H|\}$ and the cost of ϕ, $cost_{MSSC}(\phi) = \sum_{e \in E_H} \min\{\phi(v) | v \in e\}$. We let $opt_{MSSC}(H) = \min_\phi cost_{MSSC}(\phi)$ and call a solution ϕ *optimal* if $cost_{MSSC}(\phi) = opt_{MSSC}(H)$. It is known (see [4]) that a greedy algorithm solving this problem yields a 4-approximation. The algorithm in [4] works as follows. In the i-th ($i = 1, \ldots, |V_H|$) iteration a vertex v of the maximum degree in the current hypergraph is selected, $\phi(v) := i$, and all edges containing v are removed from this hypergraph. By a slight modification of the proof in [4] one can show that, if instead of selecting a vertex with the maximum degree, the procedure picks a vertex v with $d_H(v) \geq \Delta_H/2$, then the approximation ratio is nine instead of four, that is the obtained bijection ϕ satisfies

$$cost_{MSSC}(\phi) \leq 9 \cdot opt_{MSSC}(H). \tag{2}$$

Now we describe the reduction, which takes an instance $G = (V, U, E)$ with $U = \{u_1, \ldots, u_{|U|}\}$ of a semi-matching problem, and returns an MSSC instance using f-matchings. Given $f \in Z^+$, an f-matching in G is a set $Q \subseteq E$, such that $d_Q(v) \leq f$ for every $v \in V$ and $d_Q(u) \leq 1$ for every $u \in U$. Formally, given $G = (V, U, E)$ construct a hypergraph $H = H(G) = (V_H, E_H)$, where V_H is the set of all f-matchings in G and $E_H = \{e_1, \ldots, e_{|U|}\}$, where e_i is the set of all f-matchings $Q \in V_H$ with $d_Q(u_i) = 1$. Observe that, for $e_i \in E_H$, $v \in V_H$ and an f-matching Q in G corresponding to v, the fact that v belongs to e_i is equivalent to $d_Q(u_i) = 1$, and, therefore $d_H(v) = |Q|$. Moreover, the operation of removing an edge e_i from H is equivalent to removing the corresponding vertex u_i from U. As a result of this, the degrees of all vertices in e_i decrease by one and at the same time the sizes of all f-matchings containing u_i decrease by one. Notice also, that the set V_H remains unchanged during such an operation while its elements (f-matchings) might decrease in size.

The greedy algorithm finding an MSSC in H can be now rewritten as the following procedure GreedySM in G: Find a maximal f-matching Q in G, delete from U all vertices u with $d_Q(u) = 1$, and continue until G is empty. The result of GreedySM is the union M of all maximal f-matchings Q computed in the course of the procedure. Note that any maximal f-matching is a $\frac{1}{2}$-approximation of a maximum f-matching in G.

Next we relate an arbitrary semi-matching in G with a solution to the MSSC problem in $H(G)$ using a labeling. For a semi-matching M in G let ϕ_M be a solution to the MSSC problem in H defined as follows. Consider the sequence $s := 1, 1, \ldots, 1, 2, 2 \ldots, 2, 3, \ldots$ where each i appears exactly f times. Every vertex $v \in V$, in parallel, labels the edges of M incident to v by successive numbers in s. Let k be the maximum label used in this process. Then, the edges of M with label i form an f-matching Q_i. Set $\phi_M(Q_i) := i$ and for every other

f-matching Q in G let $\phi_M(Q) = j$ for some $j > k$. In addition, if M is obtained by GreedySM and the labels of edges in M correspond to iterations (edges added in the i-th iteration have label i), then ϕ_M obtained as above is said to *agree* with GreedySM.

For a semi-matching M in G, $p \in \Re$, and $v \in V$, let $k_M(v) := p \cdot d_M(v)$ if $d_M(v) < f$ and let $k_M(v) := \frac{p}{f} d_M^2(v)$ if $d_M(v) \geq f$. Let $K(M,p) := \sum_{v \in V} k_M(v)$.

Lemma 6. *For a semi-matching M, $cost_{MSSC}(\phi_M) \in [K(M,\frac{1}{2}), K(M,2)]$.*

Proof. For an edge $e = \{u,v\} \in M$, $u \in U$, let $\psi(e)$ be the label of e assigned as described before. Then $Q_{\psi(e)}$ has the smallest value of ϕ_M from all f-matchings containing u and therefore, $cost_{MSSC}(\phi_M) = \sum_{e \in M} \psi(e)$. Let $v \in V$. If $d_M(v) < f$, then the sum of $\psi(e)$ over all e's in M incident to v is $d_M(v)$ and otherwise it is $\frac{1}{2}r(r+1)f + (r+1)(d_M(v) - fr) = (r+1)(d_M(v) - \frac{1}{2}fr)$, where $r = \lfloor \frac{d_M(v)}{f} \rfloor$. Finally, we have $\frac{d_M^2(v)}{2f} \leq (r+1)(d_M(v) - \frac{1}{2}fr) \leq \frac{2d_M^2(v)}{f}$.

In the analysis of the algorithm GreedySM we use also the following fact.

Fact 4. *Let M be a semi-matching such that for all $v \in V$ it holds $d_M(v) \geq t$. Then there exists an optimal semi-matching M^* with $d_{M^*}(v) \geq t$ for all $v \in V$.*

Theorem 6. *Let $a, b \in Z^+$, $\Delta = \Delta_V(G) > ab$.* GreedySM *finds a 36-approximation of the semi-matching problem in a graph $G = (V,U,E)$ that satisfies: $d(v) \in [\Delta/a, \Delta]$ for every $v \in V$ and $d(u) \leq b$ for every $u \in U$. The algorithm runs in $O(ab^2)$ rounds.*

Proof. Let $f := \lfloor \Delta/(ab) \rfloor$ and let M be obtained by GreedySM. Let ϕ_M be the solution to the MSSC problem obtained from M that agrees with GreedySM. Then, in view of the previous discussion and by (2), $cost_{MSSC}(\phi_M) \leq 9 \cdot opt_{MSSC}(H)$. To simplify computations we redefine

$$cost_{SM}(M) := \frac{2}{f} \sum_{v \in V} d_M^2(v).$$

By Lemma 6, $\frac{1}{4} cost_{SM}(M) \leq K(M,\frac{1}{2}) \leq cost_{MSSC}(\phi_M) \leq 9 \cdot opt_{MSSC}(H)$. On the other hand, by Hall's theorem there exists a semi-matching M' with $d_{M'}(v) \geq f$ for all $v \in V$. By Fact 4 there also exists an optimal semi-matching M^* with the same property. From Lemma 6, $cost_{MSSC}(\phi_{M^*}) \leq K(M^*,2) = cost_{SM}(M^*)$ and so $opt_{MSSC}(H) \leq cost_{SM}(M^*)$. Thus, $cost_{SM}(M) \leq 36 \cdot cost_{SM}(M^*)$. The number of iterations of GreedySM is $O(ab)$ as in each iteration each vertex from V looses f or all incident edges. Finding a maximal f-matching can be done in $O(b)$ rounds (using procedures similar to those computing a maximal matching, see e.g. [13]).

Remark 1. *If $a = 1$, $b \in Z^+$, $\Delta > b$, then the algorithm* GreedySM *finds a 36-approximation of the semi-matching problem in every graph $G = (V,U,E)$, which is Δ-regular on V, in $O(b^2)$ rounds. In this case it outperforms the algorithm* SemiMatch*, which has time complexity $O(\Delta^5)$ independent of the value of b, the upper bound on the degrees of vertices in U.*

References

1. Andrasfai, B.: Introductory Graph Theory. Adam Hilger (1977)
2. Czygrinow, A., Hanćkowiak, M., Krzywdziński, K., Szymańska, E., Wawrzyniak, W.: Brief Announcement: Distributed Approximations for the Semi-matching Problem. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 200–201. Springer, Heidelberg (2011)
3. Fakcharoenphol, J., Laekhanukit, B., Nanongkai, D.: Faster Algorithms for Semi-matching Problems (Extended Abstract). In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 176–187. Springer, Heidelberg (2010)
4. Feige, U., Lovasz, L., Tetali, P.: Approximating Min Sum Set Cover. Algorithmica 40(4), 219–234 (2004)
5. Hanćkowiak, M., Karoński, M., Panconesi, A.: On the distributed complexity of computing maximal matchings. In: Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, San Francisco, CA, USA, pp. 219–225 (January 1998)
6. Harvey, N.J.A., Ladner, R.E., Lovasz, L., Tamir, T.: Semi-matchings for bipartite graphs and load balancing, J. Algorithms 59(1), 53–78 (2006)
7. Galčík, F., Katrenič, J., Semanišin, G.: On Computing an Optimal Semi-matching. In: Kolman, P., Kratochvíl, J. (eds.) WG 2011. LNCS, vol. 6986, pp. 250–261. Springer, Heidelberg (2011)
8. Linial, N.: Locality in distributed graph algorithms. SIAM Journal on Computing 21(1), 193–201 (1992)
9. Lotker, Z., Patt-Shamir, B., Rosén, A.: Distributed Approximate Matching. SIAM J. Comput. 39(2), 445–460 (2009)
10. Low, C.P.: An approximation algorithm for the load-balanced semi-matching problem in weighted bipartite graphs. Information Processing Letters 100(4), 154–161 (2006)
11. Peleg, D.: Distributed Algorithms, A Locality-Sensitive Approach. SIAM Press (2000)
12. Sadagopan, N., Singh, M., Krishnamachari, B.: Decentralized utility-based sensor network design. Mob. Netw. Appl. 11(3), 341–350 (2006)
13. Suomela, J.: Survey of Local Algorithms (manuscript), `http://www.cs.helsinki.fi/u/josuomel/doc/local-survey.pdf`

Bounds on Contention Management in Radio Networks

Mohsen Ghaffari[1], Bernhard Haeupler[1], Nancy Lynch[1], and Calvin Newport[2]

[1] Computer Science and Artificial Intelligence Lab, MIT
{ghaffari,haeupler,lynch}@csail.mit.edu
[2] Department of Computer Science, Georgetown University
cnewport@cs.georgetown.edu

Abstract. The local broadcast problem assumes that processes in a wireless network are provided messages, one by one, that must be delivered to their neighbors. In this paper, we prove tight bounds for this problem in two well-studied wireless network models: the *classical* model, in which links are reliable and collisions consistent, and the more recent *dual graph* model, which introduces unreliable edges. Our results prove that the *Decay* strategy, commonly used for local broadcast in the classical setting, is optimal. They also establish a separation between the two models, proving that the dual graph setting is strictly harder than the classical setting, with respect to this primitive.

1 Introduction

At the core of every wireless network algorithm is the need to manage contention on the shared medium. In the theory community, this challenge is abstracted as the *local broadcast problem*, in which processes are given messages, one by one, that must be delivered to their neighbors.

This problem has been studied in multiple wireless network models. The most common such model is the *classical* model, introduced by Chlamatac and Kutten [8], in which links are reliable and concurrent broadcasts by neighbors always generate collisions. The dominant local broadcast strategy in this model is the *Decay* routine introduced by Bar-Yehuda et al. [9]. In this strategy, nodes cycle through an exponential distribution of broadcast probabilities with the hope that one will be appropriate for the current level of contention (e.g., [9,11–17,22]). To solve local broadcast with high probability (with respect to the network size n), the *Decay* strategy requires $O(\Delta \log n)$ rounds, where Δ is the maximum contention in the network (which is at most the maximum degree in the network topology). It has remained an open question whether this bound can be improved to $O(\Delta + \text{polylog}(n))$. In this paper, we resolve this open question by proving the *Decay* bound optimal (notice, throughout this paper, when we call an upper bound "optimal" or a lower bound "matching," we mean within poly-$\log\log$ factors). This result also proves for the first time that existing constructions of *ad hoc selective families* [15, 16]—a type of combinatorial object used in wireless network algorithms—are optimal.

We then turn our attention to the more recent *dual graph* wireless network model introduced by Kuhn et al. [18,20,22,25]. This model generalizes the classical model by allowing some edges in the communication graph to be unreliable. It was motivated by

	Classical Model	Dual Graph Model
Ack. Upper	$O(\Delta \log n)^{**}$	$O(\Delta' \log n)^{*}$
Ack. Lower	$\Omega\left(\frac{\Delta \log n}{\log^2 \log n}\right)^{*}$	$\Omega\left(\frac{\Delta' \log n}{\log^2 \log n}\right)^{*}$
Prog. Upper	$O(\log \Delta \log n)$	$O(\min\{k \log k \log n, \Delta' \log n\})^{*}$
Prog. Lower	$\Omega(\log \Delta \log n)^{**}$	$\Omega\left(\frac{\Delta' \log n}{\log^2 \log n}\right)^{*}$

Fig. 1. A summary of our results for *acknowledgment* and *progress* for the local broadcast problem. Results that are new, or significant improvements over the previously best known result, are marked with an "*" while a "**" marks results that where obtained from prior work via minor tweaks.

the observation that real wireless networks include links of dynamic quality (see [22] for more extensive discussion). We provide tight solutions to the local broadcast problem in this setting, using algorithms based on the *Decay* strategy. Our tight bounds in the dual graph model are larger (worse) than our tight time bounds for the classical model, formalizing a separation between the two settings (see Figure 1 and the discussion below for result details). We conclude by proving another separation: in the classical model there is no significant difference in power between centralized and distributed local broadcast algorithms, while in the dual graph model the gap is exponential.

These separation results are important because most wireless network algorithm analysis relies on the correctness of the underlying contention management strategy. By proving that the dual graph model is strictly harder with respect to local broadcast, we have established that an algorithm proved correct in the classical model will not necessarily remain correct or might loose its efficiency in the more general (and more realistic) dual graph model.

To summarize: This paper provides an essentially complete characterization of the local broadcast problem in the well-studied classical and dual graph wireless network models. In doing so, we: (1) answer the long-standing open question regarding the optimality of *Decay* in the classical model; (2) provide a variant of Decay and prove it optimal for the local broadcast problem in the dual graph model; and (3) formalize the separation between these two models, with respect to local broadcast.

Result Details: As mentioned, the *local broadcast* problem assumes processes are provided messages, one by one, which should be delivered to their neighbors in the communication graph. Increasingly, local broadcast solutions are being studied separately from the higher level problems that use them, improving the composability of solutions; e.g., [18, 21, 23, 24]. Much of the older theory work in the wireless setting, however, mixes the local broadcast logic with the logic of the higher-level problem being solved; e.g., [9, 11–17, 22]. This previous work can be seen as implicitly solving local broadcast.

The efficiency of a local broadcast algorithm is characterized by two metrics: (1) an *acknowledgment bound*, which measures the time for a sender process (a process that has a message for broadcast) to deliver its message to all of its neighbors; and (2) a *progress bound*, which measures the time for a receiver process (a process that has a sender neighbor) to receive at least one message[1]. The acknowledgment bound is

[1] Note that with respect to these definitions, a process can be both a sender and a receiver, simultaneously.

obviously interesting; the progress bound has also been shown to be critical for analyzing algorithms for many problems, e.g., global broadcast [18] where the reception of *any* message is normally sufficient to advance the algorithm. The progress bound was first introduced and explicitly specified in [18, 23] but it was implicitly used already in many previous works [9, 11–14, 17]. Both acknowledgment and progress bounds typically depend on two parameters, the maximum contention Δ and the network size n. In the dual graph model, an additional measure of maximum contention, Δ', is introduced to measure contention in the unreliable communication link graph, which is typically denser than the reliable link graph. In our progress result for the dual graph model, we also introduce k to capture the *actual* amount of contention relevant to a specific message. These bounds are usually required to hold with high probability.

Our upper and lower bound results for the local broadcast problem in the classical and dual graph models are summarized in Figure 1. Here we highlight three key points regarding these results. First, in both models, the upper bounds are within $O(\log^2 \log n)$ of the lower bounds. Second, we show that $\Omega(\frac{\Delta \log n}{\log^2 \log n})$ rounds are necessary for acknowledgment in the classical model. This answers in the negative the open question of whether a $O(\Delta + \text{polylog}(n))$ solution is possible. Third, the separation between the classical and dual graph models occurs with respect to the progress bound, where the tight bound for the classical model is *logarithmic* with respect to contention, while in the dual graph model it is *linear*—an exponential gap. Finally, in addition to the results described in Figure 1, we also prove the following additional separation between the two models: in the dual graph model, the gap in progress between distributed and centralized local broadcast algorithms is (at least) linear in the maximum contention Δ', whereas no such gap exists in the classical model.

Before starting the technical sections, we remark that due to space considerations, the full proofs are omitted from the conference version and can be found in [27].

2 Model

To study the local broadcast problem in synchronous multi-hop radio networks, we use two models, namely the *classical radio network model* (also known as the radio network model) and the *dual graph model*. The former model assumes that all connections in the network are reliable and it has been extensively studied since 1980s [8–18, 18, 23]. On the other hand, the latter model is a more general model, introduced more recently in 2009 [18–20], which includes the possibility of unreliable edges. Since the former model is simply a special case of the latter, we use dual graph model for explaining the model and the problem statement. However, in places where we want to emphasize on a result in the classical model, we focus on the classical model and explain how the result specializes for this specific case.

In the dual graph model, radio networks have some reliable and potentially some unreliable links. Fix some $n \geq 1$. We define a network (G, G') to consist of two undirected graphs, $G = (V, E)$ and $G' = (V, E')$, where V is a set of n wireless nodes and $E \subseteq E'$, where intuitively set E is the set of reliable edges while E' is the set of all edges, both reliable and unreliable. In the classical radio network model, there is no unreliable edge and thus, we simply have $G = G'$, i.e., $E = E'$.

We define an algorithm $\mathcal{A}$ to be a collection of n randomized processes, described by probabilistic automata. An execution of $\mathcal{A}$ in network (G, G') proceeds as follows: first, we fix a bijection $proc$ from V to $\mathcal{A}$. This bijection assigns processes to graph nodes. We assume this bijection is defined by an adversary and is not known to the processes. We do not, however, assume that the definition of (G, G') is unknown to the processes (in many real world settings it is reasonable to assume that devices can make some assumptions about the structure of their network). In this study, to strengthen our results, our upper bounds make no assumptions about (G, G') beyond bounds on maximum contention and polynomial bounds on size of the network, while our lower bounds allow full knowledge of the network graph.

An execution proceeds in synchronous rounds $1, 2, ...$, with all processes starting in the first round. At the beginning of each round r, every process $proc(u), u \in V$ first receives inputs (if any) from the environment. It then decides whether or not to transmit a message and which message to send. Next, the adversary chooses a *reach set* that consists of E and some subset, potentially empty, of edges in $E' - E$. Note that in the classical model, set $E' - E$ is empty and therefore, the reach set is already determined. This set describes the links that will behave reliably in this round. We assume that the adversary has full knowledge of the state of the network while choosing this reach set. For a process v, let $B_{v,r}$ be the set all graph nodes u such that, $proc(u)$ broadcasts in r and $\{u, v\}$ is in the reach set for this round. What $proc(v)$ receives in this round is determined as follows. If $proc(v)$ broadcasts in r, then it receives only its own message. If $proc(v)$ does not broadcast, there are two cases: (1) if $|B_{v,r}| = 0$ or $|B_{v,r}| > 1$, then $proc(v)$ receives $\perp$ (indicating *silence*); (2) if $|B_{v,r}| = 1$, then $proc(v)$ receives the message sent by $proc(u)$, where u is the single node in $B_{v,r}$. That is, we assume processes cannot send and receive simultaneously, and also, there is no collision detection in this model. However, to strengthen our results, we note that our lower bound results hold even in the model with collision detection, i.e., where process v receives a special collision indicator message $\top$ in case $|B_{v,r}| > 1$. After processes receive their messages, they generate outputs (if any) to pass back to the environment.

Distributed vs. Centralized Algorithms: The model defined above describes distributed algorithms in a radio network setting. To strengthen our results, in some of our lower bounds we consider the stronger model of *centralized* algorithms. We formally define a centralized algorithm to be defined the same as the distributed algorithms above, but with the following two modifications: (1) the processes are given $proc$ at the beginning of the execution; and (2) the processes can make use of the current state and inputs of *all* processes in the network when making decisions about their behavior.

Notation and Assumptions: For each $u \in V$, the notations $N_G(u)$ and $N_{G'}(u)$ describe, respectively, the neighbors of u in G and G'. Also, we define $N^+_G(u) = N_G(u) \cup \{u\}$ and $N^+_{G'}(u) = N_{G'}(u) \cup \{u\}$. For any algorithm $\mathcal{A}$, we assume that each process $\mathcal{A}$ has a unique identifier. To simplify notation, we assume the identifiers are from $\{1, ..., n\}$. We remark that our lower bounds hold even with such strong identifiers, whereas for the upper bounds, we just need the identifiers of different processes to be different. Let $id(u), u \in V$ describe the id of process $proc(u)$. For simplicity, throughout this paper we often use the notation *process* u, or sometimes just u, for some $u \in V$, to refer to $proc(u)$ in the execution in question. Similarly, we sometimes use *process* i, or sometimes just i, for some $i \in \{1, ..., n\}$, to refer to the process with id i. We sometimes use the notation $[i, i']$, for integers $i' \geq i$, to indicate the sequence $\{i, ..., i'\}$, and the notation $[i]$ for integer i to indicate $[1, i]$. Throughout, we use the the notation *w.h.p.* (*with high probability*) to indicate a probability at least $1 - \frac{1}{n}$. Also, unless specified, all logarithms are natural log. Moreover, we ignore the integral part signs whenever it is clear that omitting them does not effect the calculations more than a change in constants.

3 Problem

Our first step in formalizing the local broadcast problem is to fix the input/output interface between the *local broadcast module* (automaton) of a process and the higher layers at that process. In this interface, there are three actions as follows: (1) $bcast(m)_v$, an input action that provides the local broadcast module at process v with message m that has to be broadcast over v's local neighborhood, (2) $ack(m)_v$, an output action that the local broadcast module at v performs to inform the higher layer that the message m was delivered to all neighbors of v successfully, (3) $rcv(m)_u$, an output action that local broadcast module at u performs to transfer the message m, received through the radio channel, to higher layers. To simplify definitions going forward, we assume w.l.o.g. that every $bcast(m)$ input in a given execution is for a unique m. We also need to restrict the behavior of the environment to generate $bcast$ inputs in a *well-formed* manner, which we define as strict alternation between $bcast$ inputs and corresponding ack outputs at each process. In more detail, for every execution and every process u, the environment generates a $bcast(m)_u$ input only under two conditions: (1) it is the first input to u in the execution; or (2) the last input or non-rcv output action at u was an ack.

We say an algorithm *solves the local broadcast problem* if and only if in every execution, we have the following three properties: (1) for every process u, for each $bcast(m)_u$ input, u eventually responds with a single $ack(m)_u$ output, and these are the only ack outputs generated by u; (2) for each process v, for each message m, v outputs $rcv(m)_v$ at most once and if v generates a $rcv(m)_v$ output in round r, then there is a neighbor $u \in N_{G'}(v)$ such that following conditions hold: u received a $bcast(m)_u$ input before round r and has not output $ack(m)_u$ before round r (3) for each process u, if u receives $bcast(m)_u$ in round r and respond with $ack(m)_u$ in round $r' \geq r$, then w.h.p.: $\forall v \in N_G(u)$, v generates output $rcv(m)_v$ within the round interval $[r, r']$. We call an algorithm that solves the local broadcast problem a *local broadcast algorithm*.

Time Bounds: We measure the performance of a local broadcast algorithm with respect to the two bounds first formalized in [18]: *acknowledgment* (the worst case bound on the time between a $bcast(m)_u$ and the corresponding $ack(m)_u$), and *progress* (informally speaking the worst case bound on the time for a process to receive at least one message when it has one or more G neighbors with messages to send). The first bound represents standard ways of measuring the performance of local communication. The progress bound is crucial for obtaining tight performance bounds in certain classes of applications. See [18, 23] for examples of places where progress bound proves crucial explicitly. Also, [9, 11–14, 17] use the progress bound implicitly throughout their analysis.

In more detail, a local broadcast algorithm has two *delay functions* which describe these delay bounds as a function of the relevant contention: f_{ack}, and f_{prog}, respectively. In other words, every local broadcast algorithm can be characterized by these two functions which must satisfy properties we define below. Before getting to these properties, however, we first present a few helper definitions that we use to describe local contention during a given round interval. The following are defined with respect to a fixed execution. (1) We say a process u is *active* in round r, or, alternatively, *active with* m, iff it received a $bcast(m)_u$ output in a round $\leq r$ and it has not yet generated an $ack(m)_u$ output in response. We furthermore call a message m active in round r if there is a process that is active with it in round r. (2) For process u and round r, contention $c(u, r)$ equals the number of active G' neighbors of u in r. Similarly, for every $r' \geq r$, $c(u, r, r') = max_{r'' \in [r,r']}\{c(u, r'')\}$. (3) For process v and rounds $r' \geq r$, $c'(v, r, r') = max_{u \in N_G(v)}\{c(u, r, r')\}$. We can now formalize the properties our delay functions, specified for a local broadcast algorithm, must satisfy for any execution:

1. *Acknowledgment bound:* Suppose process v receives a $bcast(m)_v$ input in round r. Then, if $r' \geq r$ is the round in which process v generates corresponding output $ack(m)_v$, then with high probability we have $r' - r \leq f_{ack}(c'(v, r, r'))$.
2. *Progress bound:* For any pair of rounds r and $r' \geq r$, and process u, if $r' - r > f_{prog}(c(u, r, r'))$ and there exists a neighbor $v \in N_G(u)$ that is active throughout the entire interval $[r, r']$, then with high probability, u generates a $rcv(m)_u$ output in a round $r'' \leq r'$ for a message m that was active at some round within $[r, r']$.

We use notation Δ' (or Δ for the classical model) to denote the maximum contention over all processes.[2] In our upper bound results, we assume that processes are provided with upper bounds on contention that are within a constant factor of Δ' (or Δ for the classical model). Also, for the sake of concision, in the results that follow, we sometimes use the terminology "*has an acknowledgment bound of*" (resp. *progress bound*) to indicate "*specifies the delay function* f_{ack}" (resp. f_{prog}). For example, instead of saying "the algorithm specifies delay function $f_{ack}(k) = O(k)$," we might instead say "the algorithm has an acknowledgment bound of $O(k)$."

Simplified One-Shot Setting for Lower Bounds: The local broadcast problem as just described assumes that processes can keep receiving messages as input forever and in an arbitrary asynchronous way. This describes the practical reality of contention management, which is an on going process. All our algorithms work in this general setting. For our lower bounds, we use a setting in which we restrict the environment to only issue broadcast requests at the beginning of round one. We call this the *one-shot setting*. Also, in most of our lower bounds, we consider, G and G' to be bipartite graphs, where nodes of one part are called *senders* and they receive broadcast inputs, and nodes of the other part are called *receivers*, and each have a sender neighbor. In this setting, when referring to contention $c(u)$, we furthermore mean $c(u, 1)$. Note that in this setting, for any r, r', $c(u, [r, r'])$ is less than or equal to $c(u, 1)$. The same holds for $c'(u)$. Also, in these bipartite networks, the maximum G'-degree (or G-degree in the classical

[2] Note that since the maximum degree in the graph is an upper bound on the maximum contention, this notation is consistent with prior work, see e.g. [18, 23, 24].

model) of the receiver nodes provides an upper bound on the maximum contention Δ' (or Δ in the classical model). When talking about these networks, and when it is clear from the context, we sometimes use the phrase *maximum receiver degree* instead of the maximum contention.

4 Related Work

Chlamatac and Kutten [8] were the first to introduce the classical radio network model. Bar-Yehuda et al. [9] studied the theoretical problem of local broadcast in synchronized multi-hop radio networks as a submodule for the broader goal of global broadcast. For this, they introduced *Decay* procedure, a randomized distributed procedure that solves the local broadcast problem. Since then, this procedure has been the standard method for resolving contention in wireless networks (see *e.g.* [17,18,23,24]). In this paper, we prove that a slightly modified version of Decay protocol achieves optimal progress and acknowledgment bounds in both the classical radio network model and the dual graph model. A summary of these time bounds is presented in Figure 1.

Deterministic solutions to the local broadcast problem are typically based on combinatorial objects called *Selective Families*, see *e.g.* [12]- [16]. Clementi et al. [14] construct (n, k)-selective families of size $O(k \log n)$ ([14, Theorem 1.3]) and show that this bound is tight for these selective families ([14, Theorem 1.4]). Using these selective families, one can get local broadcast algorithms that have progress bound of $O(\Delta \log n)$, in the classical model. These families do not provide any local broadcast algorithm in the dual graph model. Also, in the same paper, the authors construct (n, k)-strongly-selective families of size $O(k^2 \log n)$ ([14, Theorem 1.5]). They also show (in [14, Theorem 1.6]) that this bound is also, in principle, tight for selective families when $k \leq \sqrt{2n} - 1$. Using these strongly selective families, one can get local broadcast algorithms with acknowledgment bound of $O(\Delta^2 \log n)$ in the classical model and also, with acknowledgment bound of $f_{ack}(k) = O((\Delta')^2 \log n)$ in the dual graph model. As can be seen from our results (summarized in Figure 1), all three of the above time bounds are far from the optimal bounds of the local broadcast problem. This shows that when randomized solutions are admissible, solutions based on these notions of selective families are not optimal.

In [15], Clementi et al. introduce a new type of selective families called Ad-Hoc Selective Families which provide new solutions for the local broadcast problem, if we assume that processes know the network. Clementi et al. show in [15, Theorem 1] that for any given collection $\mathcal{F}$ of subsets of set $[n]$, each with size in range $[\Delta_{min}, \Delta_{max}]$, there exists an ad-hoc selective family of size $O((1 + \log(\Delta_{max}/\Delta_{min})) \cdot \log |F|)$. This, under the assumption of processes knowing the network, translates to a deterministic local broadcast algorithm with progress bound of $O(\log \Delta \, \log n)$, in the classical model. This family do not yield any broadcast algorithms for the dual graph model. Also, in [16], Clementi et al. show that for any given collection $\mathcal{F}$ of subsets of set $[n]$, each of size at most Δ, there exists a Strongly-Selective version of Ad-Hoc Selective Families that has size $O(\Delta \log |F|)$ (without using the name ad hoc). This result shows that, again under the assumption of knowledge of the network, there exists a deterministic local broadcast algorithms with acknowledgment bounds of $O(\Delta \log n)$

and $O(\Delta' \log n)$, respectively in the classical and dual graph models. Our lower bounds for the classical model show that both of the above upper bounds on the size of these objects are tight.

5 Upper Bounds for Both Classical and Dual Graph Models

In this section, we show that by slight modifications to Decay protocol, we can achieve upper bounds that match the lower bounds that we present in the next sections. Due to space considerations, the details of the related algorithms are omitted from the conference version and can be found in [27].

Theorem 5.1. *In the classical model, there exists a distributed local broadcast algorithm that gives acknowledgment bound of $f_{ack}(k) = O(\Delta \log n)$ and progress bound of $f_{prog}(k) = O(\log \Delta \log n)$.*

Theorem 5.2. *There exists a distributed local broadcast algorithm that, in the classical model, gives bounds of $f_{ack}(k) = O(\Delta \log n)$ and $f_{prog}(k) = O(\log \Delta \log n)$, and in the dual graph model, gives bounds of $f_{ack}(k) = O(\Delta' \log n)$ and $f_{prog}(k) = O(\min\{k \log \Delta' \log n, \Delta' \log n\})$.*

Theorem 5.3. *In the dual graph model, there exists a distributed local broadcast algorithm that gives acknowledgment bound of $f_{ack}(k) = O(\Delta' \log n)$ and progress bound of $f_{prog}(k) = O(\min\{k \log k \log n, \Delta' \log n\})$.*

6 Lower Bounds in the Classical Radio Broadcast Model

In this section, we focus on the problem of local broadcast in the classical model and present lower bounds for both progress and acknowledgment times. We emphasize that all these lower bounds are presented for centralized algorithms and also, in the model where processes are provided with a collision detection mechanism. Note that these points only strengthen these results. These lower bounds prove, for the first time, that the optimized decay protocol, as presented in the previous section, is optimal with respect to progress and acknowledgment times in the classical model. These lower bounds also show that the existing constructions of Ad Hoc Selective Families are optimal. Moreover, in future sections, we use the lower bound on the acknowledgment time in the classical model that we present here as a basis to derive lower bounds for progress and acknowledgment times in the dual graph model.

6.1 Progress Time Lower Bound

In this section, we remark that following the proof of the lower bound of Alon et al. [10] on the time needed for global broadcast of one message in radio networks, and with slight modifications, one can get a lower bound of $\Omega(\log \Delta \log n)$ on the progress bound in the classical model.

Lemma 6.1. *For any n and any $\Delta \leq n$, there exists a one-shot setting with a bipartite network of size n and maximum contention of at most Δ such that for any transmission schedule, it takes at least $\Omega(\log \Delta \log n)$ rounds till each receiver receives at least one message.*

6.2 Acknowledgment Time Lower Bound

In this section, we present the main technical result of the paper which is a lower bound of $\Omega(\frac{\Delta \log n}{\log^2 \log n})$ on the acknowledgment time in the classical radio broadcast model.

Theorem 6.2. *In the classical radio broadcast model, for any large enough n and any $\Delta \in [20 \log n, n^{0.1}]$, there exists a one-shot setting with a bipartite network of size n and maximum receiver degree at most Δ such that it takes at least $\Omega(\frac{\Delta \log n}{\log^2 \log n})$ rounds until all receivers have received all messages of their sender neighbors.*

In other words, in this one-shot setting, any algorithm that solves the local broadcast problem has a acknowledgment bound of $\Omega(\frac{\Delta \log n}{\log^2 \log n})$. To prove this theorem, instead of showing that randomized algorithms have low success probability, we show a stronger variant by proving an impossibility result: we prove that there exists a one-shot setting with the above properties such that, even with a centralized algorithm, it is *not possible* to schedule transmissions of nodes in $o(\frac{\Delta \log n}{\log^2 \log n})$ rounds such that each receiver receives the message of each of its neighboring senders successfully. In particular, this result shows that in this one-shot setting, for any randomized local broadcast algorithm, the probability that an execution shorter than $o(\frac{\Delta \log n}{\log^2 \log n})$ rounds successfully delivers message of each sender to all of its receiver neighbors is zero.

In order to make this formal, let us define a transmission schedule σ of length $L(\sigma)$ for a bipartite network to be a sequence $\sigma_1, \ldots, \sigma_{L(\sigma)} \subseteq S$ of senders. Having a sender $u \in \sigma_r$ indicates that at round r the sender u is transmitting its message. For a network G, we say that transmission schedule σ *covers* G if for every $v \in S$ and $u \in \mathcal{N}_G(v)$, there exists a round r such that $\sigma_r \cap \mathcal{N}_G(v) = \{u\}$, that is using transmission schedule σ, every receiver node receives all the messages of all of its sender neighbors. Also, we say that a transmission schedule σ is *short* if $L(\sigma) = o(\frac{\Delta \log n}{\log^2 \log n})$. With these notations, we are ready to state the main result of this section.

Lemma 6.3. *For any large enough n and $\Delta \in [20 \log n, n^{0.1}]$, there exists a bipartite network G with size n and maximum receiver degree at most Δ such that no short transmission schedule covers G.*

Proof (Proof Sketch for Theorem 6.3). Fix an arbitrary n and a $\Delta \in [20 \log n, n^{0.1}]$. Also let $\eta = n^{0.1}$, $m = \eta^9$. We use the probabilistic method [7] to show the existence of the network G with the aforementioned properties.

First, we present a probability distribution over a particular family of bipartite networks with maximum receiver degree Δ. To present this probability distribution, we show how to draw a random sample from it. Before getting to the details of this sampling, let us present the structure of this family. All networks of this family have a fixed set of nodes V. Moreover, V is partitioned into two nonempty disjoint sets S and R, which are respectively the set of senders and the set of receivers. We have $|S| = \eta$ and $|R| = m$. The total number of nodes in these two sets is $\eta + m = n^{0.1} + n^{0.9}$. We adjust the number of nodes to exactly n by adding enough isolated senders to the graph. To draw a random sample from this family, each receiver node $u \in R$ chooses Δ random senders from S uniformly (with replacement) as its neighbors. Also, choices of different receivers are independent of each other.

Having this probability distribution, we study the behavior of short transmission schedules over random graphs drawn from this distribution. For each fixed transmission schedule σ, let $P(\sigma)$ be the probability that σ covers a random graph G. Using a union bound, we can infer that for a random graph G, the probability that there exists a short transmission schedule that covers G is at most sum of the $P(\sigma)$-s, when σ ranges over all the short transmission schedules. Let us call this probability *the total coverage probability*. In order to prove the lower bound, we show that "the total coverage probability is in $e^{-\Omega(\eta)}$" and therefore, less than 1. Proving this claim completes the proof as with this claim, using the probabilistic method [7], we can conclude that there exists a bipartite network with maximum receiver degree of at most Δ such that no short transmission schedule covers it. To prove that the total coverage probability is $e^{-\Omega(\eta)}$, since the total number of short transmission schedules is less than 2^{η^3}, it is enough to show that for each short transmission schedule σ, $P(\sigma) = e^{-\Omega(\eta^5)}$.

Proving that for any fixed short schedule σ, $P(\sigma) = e^{-\Omega(\eta^5)}$ is the core part of the proof and also the hardest one. For this part, we use techniques similar to those that we are using in [26] for getting a lower bound for multicast in known radio networks. Let us first present some definitions. Fix a short transmission schedule σ. For each round r of σ, we say that this round is *lightweight* if $|\sigma(r)| < \frac{\eta}{2\Delta \log \eta}$. Since σ is a short transmission schedule, i.e., $L(\sigma) < \Delta \log \eta$, the total number of senders that transmit in at least one lightweight round of σ is less than $\frac{\eta}{2}$. Therefore, there are at least $\frac{\eta}{2}$ senders that never transmit in lightweight rounds of σ. We call these senders the *principal* senders of σ.

Throughout the rest of the proof, we focus on the principal senders of σ. For this, we divide the short transmission schedules into two disjoint categories, *adequate* and *inadequate*. We say that σ is an *adequate* transmission schedule if throughout σ, each principal node transmits in at least $\frac{\log \eta}{\log \log \eta}$ rounds. Otherwise we say that σ is an *inadequate* transmission schedule. We study inadequate and adequate transmission schedules in two separate lemmas (Lemmas 6.4 and 6.5), and prove that in each case $P(\sigma) = e^{-\Omega(\eta^5)}$.

Lemma 6.4. *For each inadequate short transmission schedule σ, the probability that σ covers a random graph is $e^{-\Omega(\eta^5)}$, i.e., $P(\sigma) = e^{-\Omega(\eta^5)}$.*

Proof (Proof Sketch). Let σ be an arbitrary inadequate short transmission schedule. Since σ is inadequate, there exists a principal sender node v that transmits in less than $\frac{\log \eta}{\log \log \eta}$ rounds of σ. Also, since v is a principal sender, it does not transmit in any lightweight round. That is, in each round that v transmits, the number of sender nodes that transmit is at least $\frac{\eta}{2\Delta \log \eta}$. We show that in a random graph, v is unlikely to deliver its message to all its neighbors, i.e., that in a random graph, with probability $1-e^{-\Omega(\eta^5)}$, there exists a receiver neighbor of v that does not receive the message of v.

A formal proof for this claim requires rather careful probability arguments but the main intuition is as follows. In each round that v transmits, there is a high contention, i.e., at least $\frac{\eta}{2\Delta \log \eta}$ senders transmit. Thus, in a random graph, in most of those rounds, neighbors of v receive collisions. On the other hand, the number of rounds that v transmits in them is at most $\frac{\log \eta}{\log \log \eta}$. These two observations suggest that it is unlikely for all neighbors of v to receive its message.

Lemma 6.5. *For each adequate short transmission schedule σ, the probability that σ covers a random graph is $e^{-\Omega(\eta^5)}$, i.e., $P(\sigma) = e^{-\Omega(\eta^5)}$.*

Proof (Proof Sketch). Let σ be an arbitrary adequate short transmission schedule. Recall that principal senders of σ are defined as senders that do not transmit in lightweight rounds of σ. Let us say that a message is a principal message if its sender is a principal sender. Note that in a random graph, in expectation, each receiver is adjacent to at least $\frac{\Delta}{2}$ principal senders. Therefore, if σ covers a random graph, each receiver should receive, in expectation, at least $\frac{\Delta}{2}$ principal messages. Hence, since there are m different receivers, if σ covers a random graph, there are, in expectation, at least $\frac{m\Delta}{2}$ successful deliveries. Then, using a Chernoff bound, we can infer that if σ covers a random graph, with probability $1 - e^{-\Omega(\eta^9)}$, there are at least $\frac{m\Delta}{4}$ successful deliveries. To prove the lemma, we show that the probability that for a random graph, σ has $\frac{m\Delta}{4}$ successful deliveries is $e^{-\Omega(\eta^5)}$. Then, a union bound completes the proof of lemma.

Hence, the remaining part of the proof is to show that on a random graph, with probability $e^{-\Omega(\eta^5)}$, σ has less than $\frac{m\Delta}{4}$ successful deliveries. This part is the core part of the proof of this lemma. The formal reasoning for this part requires a careful potential argument but the intuition is based on the following simple observations. Suppose that σ has at least $\frac{m\Delta}{4}$ successful deliveries with probability $e^{-\Omega(\eta^5)}$. Since σ is an adequate transmission schedule, each principal sender transmits in at least $\frac{\log \eta}{\log\log \eta}$ rounds and because there are at least $\frac{\eta}{2}$ principal senders, there has to be at least $\frac{\eta \log \eta}{2 \log\log \eta}$ transmissions by principal senders. Now in each round σ, the number of transmitting senders should be at most $\Theta(\frac{\eta}{\Delta})$, or otherwise, the number of successful deliveries drops down exponentially as a function of the multiplicative distance from $\frac{\eta}{\Delta}$, and hence the total sum of them over all the rounds would not accumulate to $\frac{m\Delta}{4}$. If we assume that in each round roughly at most $\Theta(\frac{\eta}{\Delta})$ senders transmit, we directly get a lower bound of $\frac{\frac{\eta \log \eta}{2 \log\log \eta}}{\frac{\eta}{\Delta}} = \Theta(\frac{\Delta \log \eta}{\log\log \eta})$ on the number of rounds of σ which is in contradiction with the fact that σ is short. The formal proof of this part replaces this simplistic assumption by a more careful argument that, essentially, takes all the possibilities of the number of transmitters in each of the rounds into consideration, using a potential argument. This formal argument is omitted due to the space considerations.

7 Lower Bounds in the Dual Graph Model

In this section, we show a lower bound of $\Omega(\frac{\Delta' \log n}{\log^2 \log n})$ on the progress time of centralized algorithms in the dual graph model with collision detection. This lower bound directly yields a lower bound with the same value on the acknowledgment time in the same model. Together, these two bounds show that the optimized decay protocol presented in section 5 achieves almost optimal acknowledgment and progress bounds in the dual graph model. On the other hand, this result demonstrates a big gap between the progress bound in the two models, proving that progress is unavoidably harder (slower) in the dual graph model.

Theorem 7.1. *In the dual graph model, for each n and each $\Delta' \in [20 \log n, n^{\frac{1}{11}}]$, there exists a bipartite network $H^*(n, \Delta')$ with n nodes and maximum receiver G'-degree at most Δ' such that no algorithm can have progress bound of $o(\frac{\Delta' \log n}{\log^2 \log n})$ rounds. In the same network, no algorithm can have acknowledgment bound of $o(\frac{\Delta' \log n}{\log^2 \log n})$ rounds.*

Proof (Proof Outline). In order to prove this lower bound, in Lemma 7.2, we show a reduction from acknowledgment in the bipartite networks of the classical model to the progress in the bipartite networks of the dual graph model. In particular, this means that if there exists an algorithm with progress bound of $o(\frac{\Delta' \log n}{\log^2 \log n})$ in the dual graph model, then for any bipartite network H in the classical broadcast model, we have a transmission schedule $\sigma(H)$ with length $o(\frac{\Delta \log n}{\log^2 \log n})$ that covers H. Then, we use Theorem 6.2 to complete the lower bound.

Lemma 7.2. *Consider arbitrary n_2 and Δ_2 and let $n_1 = n_2\Delta_2$ and $\Delta'_1 = \Delta_2$. Suppose that in the dual graph model, for each bipartite network with n_1 nodes and maximum receiver G'-degree Δ'_1, there exists a local broadcast algorithm A with progress bound of at most $f(n_1, \Delta'_1)$. Then, for each bipartite network H with n_2 nodes and maximum receiver degree Δ_2 in the classical radio broadcast model, there exists a transmission schedule $\sigma(H)$ with length at most $f(n_2\Delta_2, \Delta_2)$ that covers H.*

Proof (Proof Sketch). Let H be a network in the classical radio broadcast model with n_2 nodes and maximum receiver degree at most Δ_2. We use algorithm A to construct a transmission schedule σ_H of length at most $f(n_2\Delta_2, \Delta_2)$ that covers H. We first construct a new bipartite network, *Dual(H)* = (G, G'), in the dual graph model with at most n_1 nodes and maximum receiver G'-degree Δ'_1. The set of sender nodes in the Dual(H) is equal to that in H. For each receiver u of H, let $d_H(u)$ be the degree of node u in graph H. Let us call the senders that are adjacent to u 'the *associates* of u'. In the network Dual(H), we replace receiver u with $d_H(u)$ receivers and we call these new receivers 'the *proxies* of u'. In graph G of Dual(H), we match proxies of u with associates of u, i.e., we connect each proxy to exactly one associate and vice versa. In graph G' of Dual(H), we connect all proxies of u to all associates of u. It is easy to check that Dual(H) has the desired size and maximum receiver degree.

Now we present a special adversary for the dual graph model. Later we construct transmission schedule σ_H based on the behavior of algorithm A in network Dual(H) against this adversary. This special adversary activates the unreliable links using the following procedure. Consider round r and receiver node w. (1) If exactly one G'-neighbor of w is transmitting, then the adversary activates only the links from w to its G-neighbors, (2) otherwise, adversary activates all the links from w to its G'-neighbors.

We focus on the executions of algorithm A on the network Dual(H) against the above adversary. By assumption, there exists an execution α of A with length at most $f(n_2\Delta_2, \Delta_2)$ rounds such that in α, every receiver receives at least one message. Let transmission schedule σ_H be the transmission schedule of execution α. Note that because of the above choice of adversary, in the execution α, each receiver can receive messages only from its G-neighbors. Suppose that w is a proxy of receiver u of H. Then because of the construction of Dual(H), each receiver node has exactly one G-neighbor and that neighbor is one of associates of u (the one that is matched to w). Therefore, in execution α, for each receiver u of H, in union, the proxies of u receive all the messages of associates of u. On the other hand, because of the choice of adversary, if in round r of σ a receiver w receives a message, then using transmission schedule σ_H in the classical radio broadcast model, u receives the message of the same sender in round r of σ_H. Therefore, using transmission schedule σ_H in the classical broadcast model and in network H, every receiver receives messages of all of its associates. Hence, σ_H covers H and we are done with the proof of lemma.

8 Centralized vs. Distributed Algorithms in the Dual Graph Model

In this section, we show that there is a gap in power between distributed and centralized algorithms in the dual graph model, but not in the classical model—therefore highlighting another difference between these two settings. Specifically, we produce dual graph network graphs where centralized algorithms achieve $O(1)$ progress while distributed algorithms have unavoidable slow progress. In more detail, our first result shows that distributed algorithms will have *at least one process* experience $\Omega(\frac{\Delta' \log n}{\log^2 \log n})$ progress, while the second result shows the *average* progress is $\Omega(\Delta')$. Notice, such gaps do not exist in the classical model, where our distributed algorithms from Section 5 can guarantee fast progress in all networks.

Theorem 8.1. *For any k and $\Delta' \in [20 \log k, k^{1/10}]$, there exists a dual graph network of size n, $k < n \leq k^4$, with maximum receiver degree Δ', such that the optimal centralized local broadcast algorithm achieves a progress bound of $O(1)$ in this network while every distributed local broadcast algorithm has a progress bound of $\Omega(\frac{\Delta' \log n}{\log^2 \log n})$.*

Our proof argument leverages the bipartite network proven to exist in Lemma 7.2 to show that all algorithms have slow progress in the dual graph model. Here, we construct a network consisting of many copies of this counter-example graph. In each copy, we leave one of the reliable edges as reliable, but *downgrade* the others to unreliable edges that act reliable. A centralized algorithm can achieve fast progress in each of these copies as it only needs the processes connected to the single reliable edge to broadcast. A distributed algorithm, however, does not know which edge is actually reliable, so it still has slow progress. We prove that in one of these copies, the last message to be delivered comes across the only reliable edge, w.h.p. This is the copy that provides the slow progress needed by the theorem.

Notice, in some settings, practioners might tolerate a slow worst-case progress (e.g., as established in Theorem 8.1), so long as *most* processes have fast progress. In our next theorem, we show that this ambition is also impossible to achieve. To do so, we first need a definition that captures the intuitive notion of many processes having slow progress. In more detail, given an execution of the one-shot local broadcast problem (see Section 2), with processes in *sender set* S being passed messages, label each receiver that neighbors S in G with the round when it first received a message. The *average progress* of this execution is the average of these values. We say an algorithm has an *average progress of* $f(n)$, with respect to a network of size n and sender set S, if executing that algorithm in that network with those senders generates an average progress value of no more than $f(n)$, w.h.p. We now bound this metric in the same style as above

Theorem 8.2. *For any n, there exists a dual graph network of size n and a sender set, such that the optimal centralized local broadcast algorithm has an average progress of $O(1)$ while every distributed local broadcast algorithm has an average progress of $\Omega(\Delta')$.*

Our proof uses a reduction argument. We show how a distributed algorithm that achieves fast average progress in a specific type of dual graph network can be transformed to a distributed algorithm that solves global broadcast fast in a different type of dual graph network. We then apply a lower bound from [20] that proves no fast solution exists for the latter—providing our needed bound on progress.

References

1. Bachir, A., Dohler, M., Wattayne, T., Leung, K.: MAC Essentials for Wireless Sensor Networks. IEEE Communications Surveys and Tutorials 12(2), 222–248 (2010)
2. Shan, H., Zhuang, W., Wand, Z.: Distributed Cooperative MAC for Multihop Wireless Networks. IEEE Communications Magazine 47(2), 126–133 (2009)
3. Sato, N., Fujii, T.: A MAC Protocol for Multi-Packet Ad-Hoc Wireless Network Utilizing Multi-Antenna. In: Proceedings of the IEEE Conference on Consumer Communications and Networking (2009)
4. Sayed, S., and Yand, Y.: BTAC: A Busy Tone Based Cooperative MAC Protocol for Wireless Local Area Networks. In Proceedings of the Interneational Conference on Communications and Networking in China (2008).
5. Sun, Y., Gurewitz, O., Johnson, D.B.: RI-MAC: a Receiver-Initiated Asynchronous Duty Cycle MAC Protocol for Dynamic Traffic Loads in Wireless Sensor Networks. In: Proceedings of the ACM Conference on Embedded Network Sensor Systems (2008)
6. Rhee, I., Warrier, A., Aia, M., Min, J., Sichitiu, M.L.: Z-MAC: a Hybrid MAC for Wireless Sensor Networks. IEEE/ACM Trans. on Net. 16, 511–524 (2008)
7. Alon, N., Spencer, J.H.: The probabilistic method. John Wiley & Sons, New York (1992)
8. Chlamtac, I., Kutten, S.: On Broadcasting in Radio Networks–Problem Analysis and Protocol Design. IEEE Trans. on Communications (1985)
9. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the time-complexity of broadcast in radio networks: an exponential gap between determinism randomization. In: PODC 1987: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, pp. 98–108. ACM, New York (1987)
10. Alon, N., Bar-Noy, A., Linial, N., Peleg, D.: A lower bound for radio broadcast. J. Comput. Syst. Sci. 43(2), 290–298 (1991)
11. Chrobak, M., Gasieniec, L., Rytter, W.: Fast broadcasting and gossiping in radio networks. J. Algorithms 43(2), 177–189 (2002)
12. Chlebus, B.S., Gasieniec, L., Gibbons, A., Pelc, A., Rytter, W.: Deterministic broadcasting in unknown radio networks. In: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete algorithms (SODA 2000), pp. 861–870. Society for Industrial and Applied Mathematics, Philadelphia (2000)
13. Chlebus, B.S., Gasieniec, L., Östlin, A., Robson, J.M.: Deterministic Radio Broadcasting. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, p. 717. Springer, Heidelberg (2000)
14. Clementi, A., Monti, A., Silvestri, R.: Selective families, superimposed codes, and broadcasting on unknown radio networks. In: The Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 709–718. Society for Industrial and Applied Mathematics, Philadelphia (2001)

15. Clementi, A., Crescenzi, P., Monti, A., Penna, P., Silvestri, R.: On Computing Ad-hoc Selective Families. In: Proceedings of the 4th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and 5th International Workshop on Randomization and Approximation Techniques in Computer Science: Approximation, Randomization and Combinatorial Optimization, pp. 211–222 (2001)
16. Clementi, A., Monti, A., Silvestri, R.: Round robin is optimal for fault-tolerant broadcasting on wireless networks. J. Parallel Distrib. Comput. 64(1), 89–96 (2004)
17. Gasieniec, L., Peleg, D., Xin, Q.: Faster communication in known topology radio networks. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing (PODC 2005), pp. 129–137. ACM, New York (2005)
18. Kuhn, F., Lynch, N., Newport, C.: The Abstract MAC Layer. Technical Report MIT-CSAIL-TR-2009-009, MIT CSAIL, Cambridge, MA (February 20, 2009)
19. Kuhn, F., Lynch, N., Newport, C.: The Abstract MAC Layer. Distributed Computing 24(3), 187–296 (2011); Special issue from DISC 2009 23rd International Symposium on Distributed Computing
20. Kuhn, F., Lynch, N., Newport, C.: Brief Announcement: Hardness of Broadcasting in Wireless Networks with Unreliable Communication. In: Proceedings of the ACM Symposium on the Principles of Distributed Computing (PODC), Calgary, Alberta, Canada (August 2009)
21. Cornejo, A., Lynch, N., Viqar, S., Welch, J.: A Neighbor Discovery Service Using an Abstract MAC Layer. In: Forty-Seventh Annual Allerton Conference, Champaign-Urbana, IL (October 2009) (invited paper)
22. Kuhn, F., Lynch, N., Newport, C., Oshman, R., Richa, A.: Broadcasting in unreliable radio networks. In: Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2010), pp. 336–345. ACM, New York (2010)
23. Khabbazian, M., Kuhn, F., Kowalski, D.R.: Lynch. N.: Decomposing broadcast algorithms using abstract MAC layers. In: Proceedings of the 6th International Workshop on Foundations of Mobile Computing (DIALM-POMC 2010), pp. 13–22. ACM, New York (2010)
24. Khabbazian, M., Kuhn, F., Lynch, N., Medard, M., ParandehGheibi, A.: MAC Design for Analog Network Coding. In: FOMC 2011: The Seventh ACM SIGACT/SIGMOBILE International Workshop on Foundations of Mobile Computing, San Jose, CA (June 2011)
25. Censor-Hillel, K., Gilbert, S., Kuhn, F., Lynch, N., Newport, C.: Structuring Unreliable Radio Networks. In: Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, San Jose, California, June 6-8 (2011)
26. Ghaffari, M., Haeupler, B., Khabbazian, M.: The complexity of Multi-Message Broadcast in Radio Networks with Known Topology (manuscript in preparation, 2012)
27. Ghaffari, M., Haeupler, B., Lynch, N., Newport, C.: Bounds on Contention Management in Radio Networks, http://arxiv.org/abs/1206.0154

Efficient Symmetry Breaking in Multi-Channel Radio Networks

Sebastian Daum[1,⋆], Fabian Kuhn[2], and Calvin Newport[3]

[1] Faculty of Informatics, University of Lugano, Switzerland
sebastian.daum@usi.ch
[2] Department of Computer Science, University of Freiburg, Germany
kuhn@cs.uni-freiburg.de
[3] Department of Computer Science, Georgetown University, USA
cnewport@cs.georgetown.edu

Abstract. We investigate the complexity of basic symmetry breaking problems in multihop radio networks with multiple communication channels. We assume a network of synchronous nodes, where each node can be awakened individually in an arbitrary time slot by an adversary. In each time slot, each awake node can transmit or listen (without collision detection) on one of multiple available shared channels. The network topology is assumed to satisfy a natural generalization of the well-known unit disk graph model.

We study the classic *wake-up* problem and a new variant we call *active wake-up*. For the former we prove a lower bound that shows the advantage of multiple channels disappears for any network of more than one hop. For the active version however, we describe an algorithm that outperforms any single channel solution. We then extend this algorithm to compute a constant approximation for the *minimum dominating set (MDS)* problem in the same time bound. Combined, these results for the increasingly relevant multi-channel model show that it is *often* possible to leverage channel diversity to beat classic lower bounds, but not always.

1 Introduction

An increasing number of wireless devices operate in *multi-channel* networks. In these networks, a device is not constrained to use a single fixed communication channel. Instead, it can choose its channel from among the many allocated to its operating band of the radio spectrum. It can also switch this channel as needed. For example, devices using the 802.11 standard have access to around a dozen channels [1], while devices using the Bluetooth standard have access to around 75 [5].

In this paper, we prove new upper and lower bounds for symmetry breaking problems in multi-channel networks. Our goal is to use these problems to compare the computational power of this model with the well-studied *single channel* wireless model first studied by Chlamtac and Kutten [7] in the centralized setting and by Bar-Yehuda et al. [3] in the distributed setting. In more detail, we look at the *wake-up* problem [8,9,14,15], a new variant of this problem we call *active wake-up*, and the *minimum dominating set* (MDS) problem (see [16,20] for a discussion of MDS in single channel radio networks). Our results are summarized in Figure 1.

⋆ Supported by the Swiss National Science Foundation under grant n. 200021-135160.

	Single Channel	**Multi-Channel**
Single Hop Wake-Up	$\Theta(\log^2 n)$	$\mathcal{O}(\log^2 n/\mathcal{F} + \log n)$
Multihop Wake-Up	$\Omega(\log^2 n + D \log(n/D))$	$\Omega(\log^2 n + D)$
Single Hop Active Wake-Up	$\Theta(\log^2 n)$	$\mathcal{O}(\log^2 n/\mathcal{F} + \log n \log\log n)$
Multihop Active Wake-Up	$\Omega(\log^2 n + D \log n/D)$	$\mathcal{O}(\log^2 n/\mathcal{F} + \log n \log\log n)$
MDS	$\Theta(\log^2 n)$	$\mathcal{O}(\log^2 n/\mathcal{F} + \log n \log\log n)$

Fig. 1. Summary of the results we study in this paper. The *single channel* column contains the existing results from the wireless algorithm literature, though we strengthen these results model-wise in this paper. The *multi-channel* column contains our new results described for the first time (the exception is the single hop wake-up result, which derives from our recent work [11]).

Result Details and Related Work. We model a synchronous multi-channel radio network using an undirected graph $G = (V, E)$ to describe the communication topology, where G satisfies a natural geographic constraint (cf. Section 2). We assume $\mathcal{F} \geq 1$ communication channels. In each round, each node u chooses a single channel on which to participate. Concurrent broadcasts on the same channel lead to collision and there is no collision detection. For $\mathcal{F} = 1$, the model is the classical multihop radio network model [3,7].

The wake-up problem assumes that all nodes in a network begin dormant. Each dormant node can be awakened at the start of any round by an adversary. It will also awaken if a single neighbor broadcasts on the same channel. To achieve strong multi-channel lower bounds, we assume that dormant nodes can switch channels from round to round, using an arbitrary randomized strategy. To achieve strong multi-channel upper bounds, our algorithms assign the dormant nodes to a single fixed channel. In the single channel model, the best known lower bound is $\Omega(\log^2 n + D \log(n/D))$ (a combination of the $\Omega(\log^2 n)$ wake-up bound of [13,15] and the $\Omega(D \log(n/D))$ broadcast bound of [19], which holds by reduction). The best known upper bound is the near-matching $\mathcal{O}(D \log^2 n)$ randomized algorithm of [9], which generalizes the earlier single hop $\mathcal{O}(\log^2 n)$ bound of [15]. In these bounds, as with all bounds presented here, n is the network size and D the network diameter.

In Section 4.1, we prove our main lower bound result: in a multi-channel network with diameter $D > 1$, $\Omega(\log^2 n + D)$ rounds are required to solve wake-up, regardless of the size of $\mathcal{F}$. This bound holds even if we restrict our attention to networks that satisfy the strong *unit disk graph* (UDG) property.[1]

In other words, for multihop wake-up, the difficulty of the single channel and multi-channel settings are (essentially) the same. This bound might be surprising in light of our recent algorithm that solves wake-up in $\mathcal{O}(\log^2 n/\mathcal{F} + \log n)$ rounds in a multi-channel network with diameter $D = 1$ [11]. Combined, our new lower bound and the algorithm of [11] establish a gap in power between the single hop and multihop multi-channel models.

[1] Many radio network papers assume a geographic constraint on the network topology. The UDG property is arguably the strongest of these constraints. More recently, the trend has been toward looser constraints that generalize UDG (e.g., bounded independence or the clique graph constraint assumed by the algorithms in this paper).

The intuition behind our result is as follows: multiple channels help nearby awake nodes efficiently reduce contention, but they do *not* help these nodes, in a multihop setting, determine which node(s) must broadcast to awaken the dormant nodes at the next hop. The core technical idea driving this bound is a reduction from an abstract hitting game that we bound using a powerful combinatorial result proved by Alon et al. in the early 1990s [2].

In Section 4.2, we are able to leverage this same hitting game to prove a stronger version of the $\Omega(\log^2 n)$ bound of wake-up in single hop, single channel networks [13, 15]. The existing bound holds only for a restricted set of algorithms called *uniform*. Our new bound holds for general algorithms. An immediate corollary is that the $\mathcal{O}(\log^2 n)$ time, non-uniform MIS algorithm of Moscibroda and Wattenhoffer is optimal [20].

On the positive side, we consider the *active* wake-up problem, which is defined the same as the standard problem except now nodes are only activated by the adversary. The goal is to minimize the time between a node being activated and a node receiving or successfully *delivering* a message. This problem is arguably better motivated than the standard definition, as few real wireless devices are configured to allow nodes to monitor a channel and then awaken on receiving a message. The active wake-up problem, by contrast, uses activation to model a device being turned on or entering the region, and bounds the time for every device to break symmetry, not just the first device.

In a single channel network, the $\Omega(\log^2 n)$ lower bound of standard wake-up still applies to active wake-up. In Section 4.3, we describe a new algorithm that solves active wake-up in a multi-channel network in $\mathcal{O}(\log^2 n/\mathcal{F} + \log n \log\log n)$ rounds—beating the single channel lower bound for non-constant $\mathcal{F}$.

We finally turn our attention to the *minimum dominating set* (MDS) problem. In the single channel setting the $\Omega(\log^2 n)$ lower bound of [13, 15] (and our own stronger version from Section 4.2) applies via reduction. This is matched in UDGs by the $\mathcal{O}(\log^2 n)$-time MIS algorithm of [20].[2] In Section 5, we describe our main upper bound result, a $\mathcal{O}(\log^2 n/\mathcal{F} + \log n \log\log n)$-time multi-channel algorithm that also provides a constant approximation of a minimum dominating set (in expectation)—beating the single channel bounds for non-constant $\mathcal{F}$.

The key idea behind our algorithms is to leverage multi-channel diversity to filter the number of awake nodes from a potential of up to n down to $\mathcal{O}(\log^k n)$, for some constant $k \geq 1$—allowing for more efficient subsequent contention management.

Note on Proofs. Due to lack of space we sometimes only provide proof sketches rather than detailed proofs. We refer to [12] for the latter.

2 Model and Preliminaries

We model a synchronous multihop radio network with (potentially) multiple communication channels. We use an undirected graph $G = (V, E)$ to represent the communication topology for $n = |V|$ wireless *nodes*, one for each $u \in V$, and use $[\mathcal{F}] := \{1, ..., \mathcal{F}\}$, $\mathcal{F} \geq 1$, to describe the available communication channels. For each node $u \in V$ we use $N(u)$ to describe the neighbors of u in G, and let $N^k(u)$ be the set $\{v : dist(u, v) \leq k\}$.

[2] In UDGs, an MIS provides a constant-approximation of an MDS.

Nodes in our model are awakened *asynchronously*, in any round, chosen by an adversary. At the beginning of each round, each awake node u selects a channel $f \in [\mathcal{F}]$ on which to participate. It then decides to either *broadcast* a message or *receive*. A node's behavior can be probabilistic and based on its execution history up to this point. If u receives and *exactly one* node from $N(u)$ broadcasts on channel f during this round, then u receives the message, otherwise, it detects silence. If u broadcasts, it can not receive anything. That is, we assume concurrent broadcasts by neighbors on the same channel lead to collision, and there is no collision detection. Notice that u gains no direct knowledge of the behavior on other channels during this round (we assume that u only has time to tune into and receive/broadcast on a single channel per round).

When analyzing algorithms, we will assume a global round counter that starts with the first node waking up. This counter is only used for our analysis and is not known to the nodes. Furthermore, we assume nodes know n (or, a polynomial upper bound on n, which would not change our bounds), but do *not* have advanced knowledge of the network topology. In Sections 4.3 and 5, we describe algorithms in which nodes can be in many states, indicated: $\mathbb{W}$, $\mathbb{A}$, $\mathbb{C}$, $\mathbb{D}$, $\mathbb{L}$ and $\mathbb{E}$. We also use this same notation to indicate the *set* of nodes currently in that state. Finally, for ease of calculation we assume that $\log n$, $\log\log n$ and $\log n/\log\log n$ are all integers.

Graph Restrictions. When studying multihop radio networks it is common to assume some type of geographic constraint on the communication topology. In this paper, we assume a constraint that generalizes many of the constraints typically assumed in the wireless algorithms literature, including unit ball graphs with constant doubling dimension [17], which was shown in [21] to generalize (quasi) UDGs [4,18].

In more detail, let $\mathcal{R} = \{R_1, R_2, ..., R_k\}$ be a partition of the nodes in G into regions such that the sub-graph of G induced by each region R_i is a clique. The corresponding *clique graph* (or *region graph*) is a graph $G_{\mathcal{R}}$ with one node r_i for each $R_i \in \mathcal{R}$, and an edge between r_i and r_j iff $\exists u \in R_i, v \in R_j$ such that u and v are connected in G; we write $R(u)$ for the region that contains u. In this paper, we assume that G can be partitioned into cliques $\mathcal{R}$ such that the maximum degree of $G_{\mathcal{R}}$ is upper bounded by some constant parameter Δ.

Probability Preliminaries. In the following, if the probability that event A does not occur is exponentially small in some parameter k—i.e., if $\mathbf{P}(A) = 1 - e^{-ck}$ for some constant $c > 0$—we say that A happens with very high probability w.r.t. k, abbreviated as w.v.h.p.(k). We say that an event happens with high probability w.r.t. a parameter k, abbreviated as w.h.p.(k), if it happens with probability $1 - k^{-c}$, where the constant $c > 0$ can be chosen arbitrarily (possibly at the cost of adapting some other involved constants). If an event happens w.h.p.(n), we just say it happens with high probability (w.h.p.). Finally we define the abbreviation w.c.p. for *with constant probability*.

Our algorithm analysis makes use of the following lemma regarding very high probability, proved in our study of wake-up in single hop multi-channel networks [11]:

Lemma 1. *Let there be k bins and n balls with non-negative weights $w_1, \ldots, w_n \leq \frac{1}{4}$, as well as a parameter $q \in (0, 1]$. Assume that $\sum_{i=1}^{n} w_i = c \cdot k/q$ for some constant $c \geq 1$. Each ball is independently selected with probability q and each selected ball is thrown into a uniformly random bin. With probability w.v.h.p.(k), there are at least $k/4$ bins in which the total weight of all balls is between $c/3$ and $2c$.*

3 Problem

In this paper, we study two variants of the *wake-up* problem as well as the *minimum dominating set* problem. In all cases, when we say that an algorithm solves one of these problems in a certain number of rounds, then we assume this holds w.h.p.

Wake-Up: The standard definition of the wake-up problem assumes that in addition to being awakened by the adversary, a dormant node u can be awakened whenever a single neighbor broadcasts. In the multi-channel setting we assume that dormant nodes can monitor an arbitrary channel each round and they awaken if a single neighbor broadcasts on the same channel in the corresponding round. The goal of the standard wake-up problem is to minimize the time between the first and last awakening in the whole network.

Active Wake-Up: The active variant of the wake-up problem, which we are introducing in this paper, eliminates the ability for nodes to be awakened by other nodes. We instead focus on the time needed for an awaken node to *successfully* communicate with one of its neighbors. In standard wake-up dormant nodes are limited to listening only and we show that standard wake-up can be global in nature (it can take time for wake-up calls to propagate over a multihop network). The motivation for active wake-up is to have a similar problem, which allows to get past the limits imposed by the global nature of standard wake-up and still capture the most basic need within solving graph problems: communication. It turns out that active wake-up is inherently local, making it a good candidate for capturing the symmetry breaking required of local graph problems.

More formally, we say an awake node u is *completable* if at least one of its neighbors is also awake. We say a node u *completes* if it delivers a message to a neighbor or receives a message from a neighbor. The goal of active wake-up is to minimize the worst case time between a node becoming completable and subsequently completing.

Minimum Dominating Set: Given a graph $G = (V, E)$, a set $\mathbb{D} \subseteq V$ is a *dominating set* (DS) if every node in $\mathbb{E} := V \setminus \mathbb{D}$ neighbors a node in $\mathbb{D}$. A *minimum dominating set* (MDS) is a dominating set of minimum cardinality over all dominating sets for the graph. We say that a distributed algorithm solves the DS problem in time T if upon waking up, within T rounds, w.h.p., every node (irrevocably) decides to be either in $\mathbb{D}$ or in $\mathbb{E}$ such that at all times, all nodes in $\mathbb{E}$ have a neighbor in $\mathbb{D}$. We say that the algorithm computes a *constant approximation MDS* if at all times, the size of $\mathbb{D}$ is within a constant factor of the size of an MDS of the graph induced by all awake nodes.

4 Wake-Up

In this section we prove bounds on both the standard and active versions of wake-up in multi-channel networks.

4.1 Lower Bound for Standard Wake-Up

In the single channel model, there is a near tight bound of $\Omega(\log^2 n + D \log n/D)$ on the wake-up problem. We prove here that for $D > 1$ the (almost) same bound holds for multi-channel networks.

Theorem 2. *In a multi-channel network of diameter $D = 1$, the wake-up problem can be solved in $\mathcal{O}(\log^2 n/\mathcal{F} + \log n)$, but requires $\Omega(\log^2 n + D)$ rounds for $D > 1$,* *regardless of the size of $\mathcal{F}$. The lower bound holds even if we restrict our attention to network topologies satisfying the unit disk graph property.*

To better capture what makes a multihop network so difficult (and for proving Theorem 2), we reduce the following abstract game to the wake-up problem.

The Set Isolation Game. The set isolation game has a *player* face off against an adversarial *referee*. It is defined with respect to some $n > 1$ and a fixed running time $f(n)$, where f maps to the natural numbers. At the beginning of the game, the referee secretly selects a *target set* $T \subset [n]$. In each round, the player generates a *proposal* $P \subseteq [n]$ and passes it to the referee. If $|P \cap T| = 1$, the player wins and the game terminates, otherwise the referee informs the player it did not hit the set, and the game moves on to the next round without the player learning any additional information about T. If the player gets through $f(n)$ rounds without winning, it loses the game. A *strategy* $\mathcal{S}$ for the game is a randomized algorithm that uses the history of previous plays to probabilistically select the new play. We call a strategy $\mathcal{S}$ an $f(n)$ *round solution to the set isolation game*, iff for every T, w.h.p., it guarantees a win within $f(n)$ rounds.

Lemma 3. *Let $\mathcal{A}$ be an algorithm that solves wake-up in $f(n, \mathcal{F})$ rounds, for any $n > 0$ and $\mathcal{F} > 0$, when executed in a multi-channel network with diameter at least 2 and a topology that satisfies the unit disk graph property. It follows that there exists a $g_{\mathcal{F}}(n) = f(n+1, \mathcal{F})$ round solution to the set isolation game.*

Proof. Fix some $\mathcal{F}$. Our set isolation solution simulates $\mathcal{A}$ on a 2-hop network topology of size $n+1$ and with $\mathcal{F}$ channels, as follows. Let $u_1, \ldots, u_{n+1}$ be the simulated nodes. We arrange u_1 to u_n in a clique C, and connect some subset $C' \subseteq C$ to u_{n+1}. Notice, the resulting network topology satisfies the UDG property. In our simulation, the nodes in C are activated in the first round, and the player proposes, in each round of the game, the values from $[n]$ corresponding to the subset of simulated nodes $\{u_1, \ldots, u_n\}$ that broadcast during the round on the same channel chosen by u_{n+1}. (Notice, the simulator is responsible for simulating all communication and all channels.)

In this simulation, we want C' to correspond to T in the isolation game. Of course, the player simulating $\mathcal{A}$ does not have explicit knowledge of T. To avoid this problem, our simulation always simulates u_{n+1} as not receiving a message. This is valid behavior in every instance *except* for the case where exactly one node in C' broadcasts. This case, however, defines exactly when the player wins the game. If $\mathcal{A}$ isolates a single player in C' in $f(n+1, \mathcal{F})$ rounds (as is required to solve wake-up in this simulated setting), then our set isolation solution solves the set isolation game in the same time. □

To bound wake-up in multihop multi-channel networks, it is now sufficient to bound the set isolation game. Notice that bounds for a *deterministic* variant of the game could be derived from existing literature on selective families [6, 10], but we are interested here in a *randomized* solution. To obtain this bound, we leverage the following useful combinatorial result proved by Alon et al. in the early 1990s [2]:[3]

[3] Our first idea was to try to adapt the strategy used in the existing $\Omega(\log^2 n)$ bound on wake-up in single channel radio networks [13, 15]. This strategy, however, assumes a strong *uniformity* condition among the nodes, which makes sense in a single channel world—where no nodes can communicate until the problem is solved—but is too restrictive in our multi-channel world, where nodes can coordinate on the non-wake-up channels, and therefore break uniformity in their behavior.

Lemma 4 (Adapted from [2]). *Fix some $n > 0$. Let $\mathcal{H}$ and $\mathcal{J}$ be families of nonempty subsets of $[n]$. We say that $\mathcal{H}$* hits *$\mathcal{J}$ iff for every $J \in \mathcal{J}$, there is an $H \in \mathcal{H}$ such that $|J \cap H| = 1$. There exists a constant $c > 0$ and family $\mathcal{J}$, with $|\mathcal{J}|$ polynomial in n, such that for every family $\mathcal{H}$ that hits $\mathcal{J}$, $|\mathcal{H}| \geq c \log^2 n$.*

The above lemma applies to the case where there are *multiple* sets to hit, but the sets are *known* in advance. Here we translate the results to the case where there is a *single* set to hit, but the set is *unknown* in advance, and a result must hold with high probability (i.e., the exact setup of the set isolation game).

Lemma 5. *Any set isolation game strategy $\mathcal{S}$ needs $f(N) = \Omega(\log^2 N)$ rounds.*

Proof. Fix some $n > 0$. Let $N = n^k$, where $k > 1$ is a constant we fix later. Consider an execution of $\mathcal{S}$ with respect to the set $[N]$. Let $\mathcal{H}_S = (\mathcal{H}_S(r))_{1 \leq r \leq f(N)}$ be a sequence of subsets of $[n]$ such that $\mathcal{H}_S(r)$ describes the values from $[n]$ included in the proposal of $\mathcal{S}$ in round r of the execution under consideration. Let $\mathcal{J}$ be the difficult family identified by Lemma 4, defined with respect to n.

Assume for contradiction that $f(N) = o(\log^2 N)$, i.e., $f(N) < c \log^2 n$. But then, as a direct corollary of Lemma 4, there is at least one subset $J \in \mathcal{J}$ that is not hit by $\mathcal{H}_S$. With this in mind, we define the follow referee strategy for the set isolation game. Choose the target subset T from $\mathcal{J}$ uniformly at random. Any given execution of $\mathcal{S}$ fails to hit T with probability at least $1/|\mathcal{J}|$. By Lemma 4, $|\mathcal{J}|$ is polynomial in n.

Therefore, we can choose our constant k such that $1/|\mathcal{J}| > 1/n^k$.[4] It follows that the probability of failure to win the game in $f(N)$ rounds is at least $1/|\mathcal{J}| > 1/n^k = 1/N$, a contradiction to the definition of a set isolation game strategy. □

The $D > 1$ term of Theorem 2 now follows from Lemmas 3 and 5, plus a straightforward argument that $\Omega(D)$ rounds are needed to propagate information D hops, while the $D = 1$ term comes from [11].

4.2 A Stronger Single Channel Wake-Up Bound

Before continuing with our multi-channel results, we make a brief detour. By leveraging our set isolation game and Lemma 5, we can prove a stronger version of the classic $\Omega(\log^2 n)$ lower bound on wake-up in a single hop single channel network [13,15]. This existing bound holds only for *uniform* algorithms (i.e., nodes use a uniform fixed broadcast probability in each round). The version proved here holds for *general* randomized algorithms (i.e., each node's probabilistic choices can depend on its IDs and its execution history).

The argument is a variation on the simulation strategy used in Lemma 3.

Theorem 6. *Let $\mathcal{A}$ be a general randomized algorithm that solves wake-up in $f(n)$ rounds in a single hop single channel network. It follows that $f(n) = \Omega(\log^2 n)$.*

Proof. Here we follow the same general strategy exhibited by Lemma 3: showing how to use $\mathcal{A}$ to solve set isolation. Though the idea of this reduction is the same, we must alter the argument to deal with the fact that we are now in a single hop network.

[4] In the proof construction used in [2], the size $\mathcal{J}$ is bounded around n^8.

In more detail, simulate all n wake-up nodes as awake and not receiving messages. In each round, propose the set of simulated wake-up nodes that broadcast in that round. Notice, if we knew T, the obvious thing to do would be to simulate *only* the nodes corresponding to T, because by the definition of the wake-up problem, there would be a round in which exactly one of those nodes broadcasts (as required to solve wake-up). We are instead simulating all nodes. However, this does not cause a problem because each node's simulation looks the same regardless of the other nodes being simulated—in the single channel wake-up problem, nodes do not communicate with each other before the problem is solved. Consequently, for the nodes corresponding to T, this simulation is indistinguishable from one in which only these nodes were being simulated. Therefore, in some round $r \leq f(|T|) \leq f(n)$, exactly one of these nodes from T has to broadcast. The resulting proposal set will contain only one element from T (potentially in addition to some other elements from $[n] \setminus T$): solving set isolation. $\square$

The wake-up problem reduces to the MIS problem, so a bound on wake-up applies to MIS. The best known MIS algorithm for single channel radio networks is the $\mathcal{O}(\log^2 n)$-time algorithm of Moscibroda and Wattenhoffer [20]. Because their algorithm is *non-uniform*, we cannot reduce from the uniform wake-up bounds of [13, 15]. Using Theorem 6, however, the reduction now holds, proving the conjecture that the result of [20] is optimal.

4.3 Upper Bound for Active Wake-Up

In this section we present a $\mathcal{O}(\log^2 n/\mathcal{F} + \log n \log\log n)$ time solution to the active wake-up problem in a multi-channel network. For non-constant $\mathcal{F}$ this beats the $\Omega(\log^2 n)$ lower bound for this problem in the single channel setting.

Algorithm Description. Our algorithm, Algorithm 1, requires that $\mathcal{F} \geq 9$ and that $\mathcal{F} = \mathcal{O}(\log n)$ (if $\mathcal{F}$ is larger we can simply restrict ourselves to use a subset of the channels). It uses the first channel as a *competition* channel, and the remaining $\mathrm{F} = \mathcal{F} - 1$ channels for nodes in an *active* state (denoted $\mathbb{A}$). Nodes begin the algorithm in state $\mathbb{A}$, during which they choose active state channels with uniform probability and broadcast with a probability that increases exponentially from $1/n$ to $1/4$, spending only $\mathcal{O}(\log n/\mathrm{F})$ rounds at each probability. During this state, if a node receives a message it is *eliminated* ($\mathbb{E}$), at which point it receives on the competition channel for the remainder of the execution. A node that survives the active state moves on to the competition state ($\mathbb{C}$) during which it broadcasts on the competition channel with probabilities that exponentially increase from $1/\log^2 n$ to $1/2$, spending $\Theta(\log n)$ rounds at each probability. As before, receiving a message eliminates a node ($\mathbb{E}$). Finally, a node that survives the competition state advances to the leader state ($\mathbb{L}$) where it broadcasts on the competition channel with probability $1/2$ in each round.

We analyze the algorithm below.

Theorem 7. *Algorithm 1 solves the active wake-up problem in multi-channel networks in $\mathcal{O}(\log^2 n/\mathcal{F} + \log n \log\log n)$ rounds.*

As detailed in Section 2, we assume the graph can be partitioned into cliques with certain useful properties. In this proof we refer to those cliques as *regions*, which we

Algorithm 1. Active Wake-Up Algorithm

```
State description: $\mathbb{A}$ – active, $\mathbb{C}$ – competitor, $\mathbb{L}$ – leader, $\mathbb{E}$ – eliminated
begin
    $\alpha_{\mathbb{A}} = \Theta(\log n/\mathrm{F})$;    $\alpha_{\mathbb{C}} = \Theta(\log n)$
    set $count := 0$; $phase := 0$; $state := \mathbb{A}$
    while $state \neq \mathbb{E}$ do
        $count := count + 1$
        uniformly at random pick: $k \in \{2, \ldots, \mathcal{F}\}$;    $q \in [0, 1)$
        switch $state$ do
            case $\mathbb{A}$
                if $q > \frac{2^{phase}}{n}$ then listen on $k$ else send on $k$
                if $count > \alpha_{\mathbb{A}}$ then $phase := phase + 1$; $count := 0$
                if $phase > \log(n/4)$ then $phase := 0$; $state := \mathbb{C}$
            case $\mathbb{C}$
                if $q > \frac{2^{phase}}{\log^2 n}$ then listen on 1 else send on 1
                if $count > \alpha_{\mathbb{C}}$ then $phase := phase + 1$; $count := 0$
                if $phase > \log((\log^2 n)/2)$ then $state := \mathbb{L}$
            case $\mathbb{L}$
                if $q \geq 1/2$ then listen on 1 else send on 1
    Listen on 1 perpetually
Upon receiving a message:
 if $state \neq \mathbb{L}$ then $state := \mathbb{E}$
```

label $R_1, R_2, \ldots, R_k$, where $k \leq n$. We also make use of the "very high probability" notation, and corresponding Lemma 1, also presented in Section 2.

For a given round and node u, let $p(u)$ be the probability that u broadcasts in that round. Similarly, for a given round and region R, let $P_{\mathbb{A}}(R) := \sum_{u \in \mathbb{A} \cap R} p(u)$ and $P_{\mathbb{C}}(R) := \sum_{u \in (\mathbb{C} \cup \mathbb{L}) \cap R} p(u)$. When it is clear which region is meant, we sometimes omit the (R) in this notation. We begin by bounding $P_{\mathbb{A}}$ for every region R. The following lemma is a generalization of Lemma 4.8 from [11], modified to now handle a multihop network.

Lemma 8. *W.h.p., for every round and region: $P_{\mathbb{A}} = \mathcal{O}(\mathrm{F}) = \mathcal{O}(\mathcal{F})$.*

Proof Sketch. We assume that the lemma does not hold and get that in some region R the probability mass (PM) is in $\Theta(\mathrm{F})$ for the length of one phase. We can apply Lemma 1 to get $\Theta(\mathrm{F})$ channels with a $\Theta(1)$ PM each. The graph restrictions impose a limit on the amount of interference from neighboring regions. On a single such channel a successful broadcast now happens w.c.p. and it eliminates a $\Omega(1/\mathrm{F})$ fraction of the total PM. Using Chernoff we get a constant fraction reduction on the PM w.v.h.p.(F). Detailed analysis reveals that $\mathcal{O}(\log n/\mathrm{F})$ rounds are sufficient to reduce the PM by an arbitrary constant factor w.h.p., causing a contradiction. □

Lemma 9. *W.h.p., for every round and region R: $P_{\mathbb{C}} = \mathcal{O}(1)$.*

Proof Sketch. With Lemma 8 we immediately get that, w.h.p., only $\mathcal{O}(\mathrm{F}) = \mathcal{O}(\log n)$ nodes move to $\mathbb{C}$ per round and region, thus at most $\mathcal{O}(\log^2 n)$ per phase in $\mathbb{C}$. During one phase the broadcasting probability mass (PM) in one region can at most double. At the same time, continuously exceeding a certain constant threshold would imply that during one phase, w.h.p., the PM shrinks by an arbitrary constant factor. (Note that interference from neighboring regions is limited due to the graph restriction.) □

Proof (of Theorem 7). In the following, let $T = \mathcal{O}(\log^2 n/\mathcal{F} + \log n \log\log n)$ be the time required to get from waking up to $\mathbb{L}$. Consider a node u that wakes up in region R in round r. We consider two cases. In the first case, u is eliminated before it reaches $\mathbb{L}$. Therefore, u received a message in T rounds—satisfying the theorem statement.

In the second case, u reaches $\mathbb{L}$ without receiving a message. At this point T rounds have elapsed. If u is not already completable, wait until it next becomes so. Let v be the first node to make u completable. Within T rounds from waking up, v is either eliminated or in $\mathbb{C}$. In either case, it will remain on the competition channel for the remainder of the execution, where it has a chance of receiving a message from $u \in \mathbb{L}$, which would complete u. In each such round, u broadcasts with constant probability. We apply Lemma 9 to establish that the broadcast probability sum of interfering nodes (both in R and neighboring regions) is constant. Combined, u has a constant probability of delivering a message to v. For sufficiently large constant c, $c \log n$ additional rounds are sufficient for u to complete with high probability. □

5 Minimum Dominating Set

In this section, we present an algorithm that computes a constant-factor (in expectation) approximation for the MDS problem in time $\mathcal{O}(\log^2 n/\mathcal{F} + \log n \log\log n)$. For $\mathcal{F} = \omega(1)$ this outperforms the fastest known algorithm to solve MDS in the single channel model. For $\mathcal{F} = \mathcal{O}(\log n/\log\log n)$ the speed-up is in the order of $\Theta(\mathcal{F})$.

Algorithm Description. Algorithm 2 builds on the ideas of the active wake-up algorithm of the previous section as follows. For simplicity, we assume that $\mathcal{F} = \mathcal{O}(\log n)$, as more frequencies are not exploited. For an easier handling of the analysis we partition and rename the $\mathcal{F}$ available channels $[\mathcal{F}]$ into $\{\mathcal{A}_1, \ldots, \mathcal{A}_{\mathrm{F}}\} \dot\cup \{\mathcal{D}_1, \ldots, \mathcal{D}_{n_{\mathcal{D}}}\} \dot\cup \{\mathcal{C}\}$, such that $\mathrm{F} = \Theta(\mathcal{F})$ and $n_{\mathcal{D}} = \mathcal{O}(\min\{\log\log n, \mathcal{F}\})$.

After being woken up, a node u starts in the *waiting state* $\mathbb{W}$, in which it listens uniformly at random on channels $\mathcal{D}_1, \ldots, \mathcal{D}_{n_{\mathcal{D}}}$. Its goal is to hear from a potentially already existing neighboring dominator before it moves on to the *active state* $\mathbb{A}$. Once in $\mathbb{A}$ node u starts broadcasting on the channels $\{\mathcal{A}_1, \ldots, \mathcal{A}_{\mathrm{F}}\}$ with probability $1/n$ in each round. It acts in phases and at the beginning of each phase it doubles its broadcasting probability until it reaches probability $1/4$. As in the wake-up protocol, u chooses its channel uniformly at random, allowing us to reduce the length of each phase from the usual $\Theta(\log n)$ in a single channel setting to $\Theta(\log n/\mathrm{F})$, while still keeping the broadcasting probability mass in each region bounded w.h.p.

Unlike the wake-up algorithm, a node is not done when it receives a message. Thus, if a node receives a message in state $\mathbb{A}$ then it restarts with state $\mathbb{W}$. If a node manages to broadcast in state $\mathbb{A}$, it immediately moves on to the *candidate state* $\mathbb{C}$. Because the probability mass in $\mathbb{A}$ is bounded in every region, the number of nodes moving to the candidate state can also be bounded by $\mathcal{O}(\operatorname{polylog} n)$.

State $\mathbb{C}$ starts with a long *sleeping phase* (phase 0) in which nodes act as in state $\mathbb{W}$, i.e., they listen on channels $\mathcal{D}_1, \ldots, \mathcal{D}_{n_\mathcal{D}}$: to find out about potential dominators created while they were in state $\mathbb{A}$. If a node u does not receive the message of a neighboring dominator in that time it moves on to phases $1, 2, \ldots$, during which u tries to become a dominator by broadcasting on channel $\mathcal{C}$. Unlike in state $\mathbb{A}$, u can start with broadcasting probability $1/\log^2 n$, without risk of too much congestion. This allows us to reduce the total number of phases to $\mathcal{O}(\log\log n)$. A candidate that manages to broadcast, immediately moves on to the *dominating state* $\mathbb{D}$, while candidates receiving a message from another candidate move to the *eliminated state* $\mathbb{E}$, because they know that the sender of that message is now a dominator. Assuming that $\mathcal{F} = \Omega(\log\log n)$, dominators run the following protocol. In each round, they choose a channel $\mathcal{D}_i$ uniformly at random and broadcast on it with probability 2^{-i}. We can show that the number of dominators in each node v's neighborhood is at most poly-logarithmic in n. Then, as soon as v has at least one dominator in its neighborhood, there is always a channel $\mathcal{D}_\lambda$ on which v can receive a message from a dominator with constant probability. On average v will choose the right channel within $\mathcal{O}(\log\log n)$ rounds, so $\mathcal{O}(\log n \log\log n)$ rounds are enough to ensure high probability. In the case $\mathcal{F} = o(\log\log n)$ a constant number of channels with appropriate broadcasting probabilities suffice to make a dominator heard within $\mathcal{O}(\log^2 n/\mathcal{F} + \log n \log\log n)$ rounds.

We analyze the algorithm below.

Theorem 10. *Algorithm 2 computes a constant approximation for the MDS problem in time* $\mathcal{O}(\log^2 n/\mathcal{F} + \log n \log\log n)$.

Let us start out with some definitions and notations. We define $P_\mathbb{A}$ and $P_\mathbb{C}$ analogously to Section 4.3. Further, we call a node *decided* if it belongs to $\mathbb{D}$ or $\mathbb{E}$. A region R is called *decided* in round r, if no node in R is in $\mathbb{A}$ or $\mathbb{C}$ in any round $r' \geq r$. Hence, in particular after a region R becomes decided, no dominators will be created in R. Finally, we define $T' := \alpha_\mathbb{W} + \alpha_{sleep} + (\alpha_\mathbb{A} + \alpha_\mathbb{C} + 2)\log n$ and $T := 3(\Delta^2 + 1)T'$.

Lemma 11. *W.h.p., at all times and for every region* R*, the probability mass* $P_\mathbb{A}$ *in* R *is bounded by* $\mathcal{O}(\mathrm{F})$.

Proof. The proof is identical to the proof of Lemma 8 for the wake-up algorithm. □

Lemma 12. *W.h.p., at most* $\mathcal{O}(\mathrm{F} + \log n) = \mathcal{O}(\log n)$ *nodes switch to the candidate phase in any region* R *in any round* r.

Proof. By Lemma 11, w.h.p., the probability mass $P_\mathbb{A}$ is always bounded by $c\mathrm{F}$ for some constant c. For each node v in the region R, define X_v as the Bernoulli random variable that indicates whether v moves to the candidate phase in round r and let $X := \sum_{v \in R} X_v$ and $\mu := \mathbf{E}[X] \leq P_\mathbb{A} \leq c\mathrm{F}$. For an arbitrary $d > 0$ let $\delta := \mu^{-1}(e^2 c\mathrm{F} + d\log n) - 1$. Then, applying a standard Chernoff bound, we get

$$\mathbf{P}(X \geq (1+\delta)\mu) = \mathbf{P}(X \geq (e^2 c\mathrm{F} + d\log n)) \leq e^{-\mu(\delta+1)} \leq e^{-d\log n} = n^{-d}. \qquad \square$$

Algorithm 2. Dominating Set Algorithm

```
States:    $\mathbb{W}$ – waiting, $\mathbb{A}$ – active, $\mathbb{C}$ – candidate, $\mathbb{D}$ – dominator, $\mathbb{E}$ – eliminated
Channels:  $\mathcal{A}_1, \dots, \mathcal{A}_{\mathrm{F}}$ – filtering, $\mathcal{D}_1, \dots, \mathcal{D}_{n_{\mathcal{D}}}$ – notification, $\mathcal{C}$ – competition
begin
    $\alpha_{\mathbb{W}} = \alpha_{sleep} = \Theta(\log^2 n/\mathcal{F} + \log n \log\log n)$;  $\alpha_{\mathbb{A}} = \Theta(\log n/\mathrm{F})$;  $\alpha_{\mathbb{C}} = \Theta(\log n)$
    set $count := 0$; $state := \mathbb{W}$
    if $\mathcal{F} = \Omega(\log\log n)$ then $n_{\mathcal{D}} := \Theta(\log\log n)$ else $n_{\mathcal{D}} := 4$
    while $state \neq \mathbb{E}$ do
        $count := count + 1$
        uniformly at random pick:  $i \in \{1, \dots, n_{\mathcal{D}}\}$;  $k \in \{1, \dots, \mathrm{F}\}$;  $q \in [0, 1)$
        switch state do
            case $\mathbb{W}$
                listen on $\mathcal{D}_i$
                if $count = \alpha_{\mathbb{W}}$ then $count := 0$, $state := \mathbb{A}$, $phase := 0$
            case $\mathbb{A}$
                if $count = \alpha_{\mathbb{A}}$ then $count := 0$, $phase := \min\{phase + 1, \log(n/4)\}$
                if $q > \frac{2^{phase}}{n}$ then listen on $\mathcal{A}_k$
                else send on $\mathcal{A}_k$; $count := 0$, $phase := 0$, $state := \mathbb{C}$
            case $\mathbb{C}$
                if $phase = 0$ then
                    listen on $\mathcal{D}_i$
                    if $count = \alpha_{sleep}$ then $count := 0$, $phase := 1$
                else
                    if $count = \alpha_{\mathbb{C}}$ then $count := 0$, $phase := \min\{phase + 1, 2\log\log n\}$
                    if $q > \frac{2^{phase-1}}{\log^2 n}$ then listen on $\mathcal{C}$ else send on $\mathcal{C}$; $state := \mathbb{D}$
            case $\mathbb{D}$
                if $n_{\mathcal{D}} = 4$ then $p := \left(\frac{\mathcal{F}}{\log n}\right)^i$ else $p := 2^{-i}$
                with probability $p$ send on $\mathcal{D}_i$
Upon receiving a message:
 if $state = \mathbb{A}$ then $count := 0$, $state := \mathbb{W}$ else $state := \mathbb{E}$
```

Lemma 13. *W.h.p., at all times and for every region R, the probability mass $P_{\mathbb{C}}$ in R is bounded by $\mathcal{O}(1)$.*

Proof. By Lemma 12, in no round more than $\mathcal{O}(\log n)$ nodes move from state $\mathbb{A}$ to state $\mathbb{C}$. Thus at most $\mathcal{O}(\log^2 n)$ nodes do so within the length $\alpha_{\mathbb{C}}$ of one phase (not phase 0) of state $\mathbb{C}$. The claim then follows analogously to the proof of Lemma 9. □

The purpose of the sleeping phases in state $\mathbb{W}$ and at the beginning of state $\mathbb{C}$ is for nodes to detect if they have a dominator in their neighborhood and thus getting eliminated before going to $\mathbb{A}$ or to start competing in $\mathbb{C}$. The following lemma shows that both sleep phases do their job and that a full sleep phase is enough for a dominator to eliminate a neighbor in state $\mathbb{W}$ or phase 0 of state $\mathbb{C}$.

Lemma 14. *Assume that a node u starts with state $\mathbb{W}$ or phase 0 of state $\mathbb{C}$ in round r and there is already a dominator in $N(u)$. Further, assume that $k_t := |\mathbb{D} \cap (N^1(u))| = \mathcal{O}(\log^3 n/\mathcal{F} + \log^2 n \log\log n)$ at all times $t \in [r, r + \alpha_{\mathbb{W}}] = [r, r + \alpha_{sleep}]$. Then, w.h.p., u switches to state $\mathbb{E}$ by round $r + \alpha_{\mathbb{W}} = r + \alpha_{sleep}$.*

Proof Sketch. First assume that $n_{\mathcal{D}} = \Omega(\log\log n)$. Because of k_t being bounded as demanded there is a 'favored' channel $\mathcal{D}_{\lambda_t}$ on which the broadcasting probability mass (PM) is in $\Theta(1)$. Thus, u has a $\Theta(1/\log\log n)$ probability to hit that channel each round. α_{sleep} rounds suffice for u to receive a message w.h.p. If $n_{\mathcal{D}} = o(\log\log n)$, then 4 channels suffice. The stated probabilities in Algorithm 2 ensure that on one of the 4 channels the PM is in $\Omega(\mathcal{F}/\log n) \cap \mathcal{O}(1)$, i.e., $\mathcal{O}(\log n/\mathcal{F})$ rounds per phase are enough for receiving a message w.h.p. □

The following lemma shows that the number of dominators in each region is bounded and that as soon as there is a dominator in a region, the region also becomes decided within bounded time.

Lemma 15. *The lemma statement is in three parts:*

(a) W.h.p., in every region R and round r: only $\mathcal{O}(\log n)$ nodes move to state $\mathbb{D}$.
(b) W.h.p., if there is a node u in state $\mathbb{D}$, then all nodes that are already awake in $N^1(u) \supset R(u)$ are decided within time T'.
(c) W.h.p., in every region R: $|\mathbb{D}| = \mathcal{O}(\log^3 n/\mathcal{F} + \log^2 n \log\log n)$.

Proof Sketch. Part *(a)* is proven similar to Lemma 12 using the result of Lemma 13.

For *(b)* let $v \in N(u)$. Due to *(a)* there are not too many dominators created in $N(v)$, so there is no congestion on channels $\mathcal{D}_1, \ldots, \mathcal{D}_{n_{\mathcal{D}}}$. Lemma 14 then provides that nodes in states $\mathbb{A}$ or $\mathbb{W}$ will get eliminated soon. Nodes in state $\mathbb{C}$ will be decided eventually due to the algorithm construction.

Part *(c)* follows from combining *(a)*, *(b)* and Lemma 14. □

Lemma 16. *W.h.p., each node u that wakes up is decided within $T = \mathcal{O}(\log^2 n/\mathcal{F} + \log n \log\log n)$ rounds.*

Proof Sketch. Note that if u leaves states $\mathbb{W}$ and $\mathbb{A}$ behind then it will get decided within T' rounds. Thus, for not getting decided soon it has to be set back to state $\mathbb{W}$ often. But every time this happens, a node $v \in N(u)$ moves to state $\mathbb{C}$, getting decided eventually, implying the creation of a dominator $w \in N^1(v) \subset N^2(u)$ within T' rounds. Lemma 15 limits the time until $R(w)$ is decided. Finally, this can happen at most $\Delta^2 + 1 = \mathcal{O}(1)$ times. □

Lemma 17. *For each region R, the expected number of nodes that become dominators in region R is bounded by $\mathcal{O}(1)$.*

Proof. Consider some fixed region R and let t_0 be the first time when a node becomes a candidate in region R. For $i = 1, 2, \ldots$, let $P_{\mathbb{C},i}$ be the sum of the broadcast probabilities of all the candidates in region R on channel $\mathcal{C}$ in round $t_0 + i$. As nodes have to be candidates before becoming dominators, dominators in region R can only be created after time t_0. Let X_i be the number of dominators created in R in round $t_0 + i$ and let $X = \sum_{i\geq 1} X_i$. To prove the lemma, we have to show that $\mathbf{E}[X] = \mathcal{O}(1)$.

We say that a newly created dominator v in round r *clears* its region R iff v is the only dominator created in region R in round r and all candidates in region R hear v's message on channel $\mathcal{C}$ in round r. Clearly, all nodes that are candidates in region R in round r switch to state $\mathbb{E}$ when this occurs. Therefore, the only nodes in R that can still become dominators must either be in another state ($\mathbb{W}$, $\mathbb{A}$) or not yet awake. By Lemma 15, $|\mathbb{D} \cap R|$ is always bounded such that by Lemma 14, w.h.p., the latter nodes also do not become dominators.

Having established the power of clearing, we bound the probability of such events. In more detail, let $\mathcal{E}_i$ be the event that in round t_0+i some node v in region R becomes a dominator by clearing R. We next show that such a clearance happens with probability at least $\delta \cdot P_{\mathbb{C},i}$ for some constant $\delta > 0$. To see why, recall that by Lemma 13, we know that for all $i \geq 1$, $P_{\mathbb{C},i}$ as well as $P_{\mathbb{C},i}(R')$ in round $t_0 + i$ for every neighboring region R' are upper bounded by some constant $\hat{P}_{\mathbb{C}}$. For each candidate v in region R let $p(v)$ be its broadcasting probability. Then the probability that exactly one candidate from region R broadcasts on channel $\mathcal{C}$ in round $t_0 + i$ is lower bounded by

$$\sum_{v\in R\cap\mathbb{C}} p(v) \prod_{u\in R\cap\mathbb{C},\, u\neq v} (1 - p(u)) \geq P_{\mathbb{C},i} 4^{-\hat{P}_{\mathbb{C}}} = \Omega(P_{\mathbb{C},i}).$$

The probability that no candidate from any neighboring region R' (of which there are at most Δ) broadcasts on channel $\mathcal{C}$ in round $t_0 + i$ is at least $4^{-\Delta\hat{P}_{\mathbb{C}}} = \Omega(1)$. Hence, there exists a constant $\delta > 0$ such that $\mathbf{P}(\mathcal{E}_i) \geq \delta P_{\mathbb{C},i}$.

In the following, we define $Q_i := \sum_{j=1}^{i} P_{\mathbb{C},i}$. The probability that no node v clears region R by some time $t_0 + \tau$ can be upper bounded by

$$\mathbf{P}\left(\bigcap\nolimits_{i=1}^{\tau} \overline{\mathcal{E}_i}\right) \leq \prod\nolimits_{i=1}^{\tau} (1 - \delta P_{\mathbb{C},i}) < e^{-\delta \sum_{i=1}^{\tau} P_{\mathbb{C},i}} = e^{-\delta Q_\tau}.$$

As discussed above, w.h.p., a clearance in R prevents new nodes from subsequently becoming dominators in this region. Let $\mathcal{G}$ be the event that this high probability property holds. When we condition on $\mathcal{G}$, it holds that a dominator can join a region in a given round only if there have been no previous clearances in that region. Hence,

$$\mathbf{E}\left[X_i|\mathcal{G}\right] \leq \mathbf{P}\left(\bigcap\nolimits_{j=1}^{i-1} \overline{\mathcal{E}}_i\right) \cdot P_{\mathbb{C},i}$$

and therefore

$$\mathbf{E}\left[X|\mathcal{G}\right] \leq \sum\nolimits_{i\geq 1} P_{\mathbb{C},i} \cdot e^{-\delta Q_{i-1}} = \mathcal{O}(1).$$

Because $\mathcal{G}$ happens w.h.p., and $|\mathbb{D}| \leq n$, we get $\mathbf{E}[X] = \mathcal{O}(1)$. □

Proof (of Theorem 10). By Lemma 16, w.h.p., after it wakes up every node is decided within $\mathcal{O}(\log^2 n/\mathcal{F} + \log n \log\log n)$ rounds. Since a node only goes to state $\mathbb{E}$ after hearing from a neighboring dominator, the computed dominating set is valid. Finally, by Lemma 17, in expectation, the algorithm computes a constant approximation of the optimal MDS solution. □

References

1. I. 802.11. Wireless LAN MAC and Physical Layer Specifications (June 1999)
2. Alon, N., Bar-Noy, A., Linial, N., Peleg, D.: A Lower Bound for Radio Broadcast. Journal of Computer and System Sciences 43(2), 290–298 (1991)
3. Bar-Yehuda, R., Goldreich, O., Itai, A.: On the Time-Complexity of Broadcast in Multi-Hop Radio Networks: An Exponential Gap Between Determinism and Randomization. Journal of Computer and System Sciences 45(1), 104–126 (1992)
4. Barriére, L., Fraigniaud, P., Narayanan, L.: Robust position-based routing in wireless ad hoc networks with unstable transmission ranges. In: Proc. 5th Int. Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIALM), pp. 19–27 (2001)
5. Bluetooth Consortium. Bluetooth Specification Version 2.1 (July 2007)
6. Bonis, A.D., Gasieniec, L., Vaccaro, U.: Generalized fFamework for Selectors with Applications in Optimal Group Testing. In: Proceedings of the International Colloquium on Automata, Languages and Programming (2003)
7. Chlamtac, I., Kutten, S.: On Broadcasting in Radio Networks–Problem Analysis and Protocol Design. IEEE Transactions on Communications 33(12), 1240–1246 (1985)
8. Chlebus, B.S., Kowalski, D.R.: A Better Wake-Up in Radio Networks. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, pp. 266–274. ACM (2004)
9. Chrobak, M., Gasieniec, L., Kowalski, D.: The Wake-Up Problem in Multi-Hop Radio Networks. In: Proceedings of the Annual Symposium on Discrete Algorithms, pp. 992–1000. Society for Industrial and Applied Mathematics (2004)
10. Clementi, A.E.F., Monti, A., Silvestri, R.: Distributed Broadcast in Radio Networks of Unknown Topology. Theoretical Computer Science 302(1-3) (2003)
11. Daum, S., Gilbert, S., Kuhn, F., Newport, C.: Leader Election in Shared Spectrum Networks. In: Proceedings of the Principles of Distributed Computing (to appear, 2012)
12. Daum, S., Kuhn, F., Newport, C.: Efficient Symmetry Breaking in Multi-Channel Radio Networks. Technical Report 271, University of Freiburg, Dept. of Computer Science (2012)
13. Farach-Colton, M., Fernandes, R.J., Mosteiro, M.A.: Lower Bounds for Clear Transmissions in Radio Networks. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 447–454. Springer, Heidelberg (2006)
14. Gasieniec, L., Pelc, A., Peleg, D.: The Wakeup Problem in Synchronous Broadcast Systems. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, pp. 113–121 (2000)
15. Jurdzinski, T., Stachowiak, G.: Probabilistic Algorithms for the Wakeup Problem in Single-Hop Radio Networks. In: Proceedings of the International Symposium on Algorithms and Computation, pp. 535–549 (2002)
16. Kuhn, F., Moscibroda, T., Wattenhofer, R.: Initializing Newly Deployed Ad Hoc and Sensor Networks. In: Proceedings of International Conference on Mobile Computing and Networking, pp. 260–274. ACM (2004)
17. Kuhn, F., Moscibroda, T., Wattenhofer, R.: On the locality of bounded growth. In: Proc. 24th Symp. on Principles of Distributed Computing (PODC), pp. 60–68 (2005)
18. Kuhn, F., Wattenhofer, R., Zollinger, A.: Ad hoc networks beyond unit disk graphs. Wireless Networks 14(5), 715–729 (2008)
19. Kushilevitz, E., Mansour, Y.: An (D log (N/D)) Lower Bound for Broadcast in Radio Networks. SIAM Journal on Computing 27(3), 702–712 (1998)
20. Moscibroda, T., Wattenhofer, R.: Maximal independent sets in radio networks. In: Proc. 24th Symp. on Principles of Distributed Computing (PODC), pp. 148–157 (2005)
21. Schmid, S., Wattenhofer, R.: Algorithmic models for sensor networks. In: Proc. 14th Int. Workshop on Parallel and Distributed Real-Time Systmes, pp. 1–11 (2006)

On Byzantine Broadcast in Loosely Connected Networks

Alexandre Maurer and Sébastien Tixeuil

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

Abstract. We consider the problem of reliably broadcasting information in a multihop asynchronous network that is subject to Byzantine failures. Most existing approaches give conditions for perfect reliable broadcast (all correct nodes deliver the authentic message and nothing else), but they require a highly connected network. An approach giving only probabilistic guarantees (correct nodes deliver the authentic message with high probability) was recently proposed for loosely connected networks, such as grids and tori. Yet, the proposed solution requires a specific initialization (that includes global knowledge) of each node, which may be difficult or impossible to guarantee in self-organizing networks – for instance, a wireless sensor network, especially if they are prone to Byzantine failures.

In this paper, we propose a new protocol offering guarantees for loosely connected networks that does not require such global knowledge dependent initialization. In more details, we give a methodology to determine whether a set of nodes will always deliver the authentic message, in any execution. Then, we give conditions for perfect reliable broadcast in a torus network. Finally, we provide experimental evaluation for our solution, and determine the number of randomly distributed Byzantine failures than can be tolerated, for a given correct broadcast probability.

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

1 Introduction

In this paper, we study the problem of reliably broadcasting information in a network that is subject to attacks or failures. Those are an important issue in a context where networks grow larger and larger, making the possibility of failure occurrences more likely. Many models of failures and attacks have been studied so far, but the most general model is the *Byzantine* model [11]: some nodes in the network may exhibit arbitrary behavior. In other words, all possible behaviors must be anticipated, including the most malicious strategies. The generality of this model encompasses a rich panel of security applications.

In the following, we assume that a correct node (the *source*) broadcasts a message in a network that may contain Byzantine nodes. We say that a correct node *delivers* a message, when it considers that this actually is the message broadcasted by the source.

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 weakens the power of Byzantine nodes, as they ignore some cryptographic secrets: their behavior is not totally arbitrary. Moreover, in some applications such as sensor networks, the nodes may not have enough resources to manipulate digital signatures. Finally, cryptographic operations require the presence of a trusted infrastructure, such as secure channels to a key server or a public key infrastructure. In this paper, we focus on non-cryptographic and totally distributed solutions: no element of the network is more important than another, and all elements are likely to fail.

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 partially connected networks, where a node must rely on other nodes to broadcast informations.

Dolev [4] considers Byzantine agreement on arbitrary graphs, and states 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 the number of nodes in the system is at least $3k+1$. Also, this solution assumes that the topology is known to every node, and that nodes are scheduled according to the synchronous execution model. Nesterenko and Tixeuil [19] relax both requirements (the topology is unknown and the scheduling is asynchronous) yet retain $2k+1$ connectivity for resilience 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. More precisely, if there are two ore more Byzantine nodes *anywhere* in the grid, there always exists a possible execution where no correct node delivers the authentic message.

Byzantine-resilient broadcast was also investigated in the context of *radio networks*: each node is a robot or a sensor with a physical position. A node can only communicate with nodes that are located within a certain radius. Broadcast protocols have been proposed [10,2] for nodes organized on a grid. However, the wireless medium typically induces much more than four neighbors per node, otherwise the broadcast does not work (even if all nodes are correct). Both approaches are based on a local voting system, and perform correctly if every node has less than a $1/4\pi$ fraction of Byzantine neighbors. This criterion was later generalized [20] to other topologies, assuming that each node knows the global topology. Again, in loosely connected networks, the local constraint on the proportion of Byzantine nodes in any neighborhood may be difficult to assess.

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.

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 connectivity, which can be a heavy constraint in a large network. To overcome this problem, a probabilistic approach for reliable broadcast has been proposed in [16]. In this setting, the distribution of Byzantine failures is assumed to be random. This hypothesis is realistic in various networks such as a peer-to-peer overlays, where the nodes joining the network are not able to choose their localization, and receive a randomly generated identifier that determines their location in the overlay. Also, it is considered acceptable that a small minority of correct nodes are fooled by the Byzantine nodes. With these assumptions, the network can tolerate [16] a number of Byzantine failures that largely exceeds its connectivity. Nevertheless, this solution requires to define many sets of nodes (called *control zones* [16]) before running the protocol: each node must initially know to which control zones it belongs. This may be difficult or impossible in certains types of networks, such as a self-organized wireless sensor network or a peer-to-peer overlay.

Our Contribution. In this paper, we propose a broadcast protocol performing in loosely connected networks subject to Byzantine failures that relaxes the aforementioned constraint – no specific initialization is required for the nodes. This protocol is described in Section 2. Further, we prove general properties on this protocol, and use them to give both deterministic and probabilistic guarantees.

In Section 3, we give a sufficient condition for *safety* (no correct node delivers a false message). This condition is not based on the number, but on the *distance* (with respect to the number of hops) between Byzantine failures. Then, we give a methodology to construct – node by node – a set of correct nodes that will always deliver the authentic message, in any possible execution.

In Section 4, we consider a particular loosely connected network: the *torus*, where each node has exactly four neighbors. We give a sufficient condition to achieve perfect reliable broadcast on such a network (all correct nodes deliver the authentic message).

In Section 5, we make an experimental evaluation of the protocol on *grid* networks. We give a methodology to estimate the probability that a correct node delivers the authentic message, for a given number of Byzantine failures. This way, we can determine the maximal number of failures that the network can hold, to achieve a given probabilistic guarantee.

2 Description of the Protocol

In this section, we provide an informal description of the protocol. Then, we precise our notations and hypotheses, and give the algorithm that each correct node must follow.

2.1 Informal Description

The network is described by a set of processes, called *nodes*. Some pairs of nodes are linked by a *channel*, and can send messages to each other: we call them *neighbors*. The network is *asynchronous*: the nodes can send and receive messages at any time.

A particular node, called the *source*, wants to broadcast an information m to the rest of the network. In the ideal case, the source would send m to its neighbors, which will transmit m to their own neighbors – and so forth, until every node receives m. In our setting however, some nodes – except the source — can be malicious (*Byzantine*) and broadcast false informations to the network. Of course, a correct node cannot know whether a neighbor is Byzantine.

To limit the diffusion of false messages, we introduce a *trigger* mechanism: when a node p receives a message m, it must wait the reception of a *trigger message* to accept and retransmit m. The *trigger message* informs p that another node, located at a reasonable distance, has already accepted m. This distance is the number H of channels (or *hops*) that the trigger message can cross. This is illustrated in Figure 1-a.

The underlying idea is as follows: if the Byzantine nodes are sufficiently spaced, they will never manage to broadcast false messages. Indeed, to broadcast a false message, a Byzantine node requires an accomplice to broadcast the corresponding trigger message (see Figure 1-b). However, if this accomplice is distant from more than $H + 1$ hops, the trigger message will never reach its target, and the false message will never be accepted (see Figure 1-c).

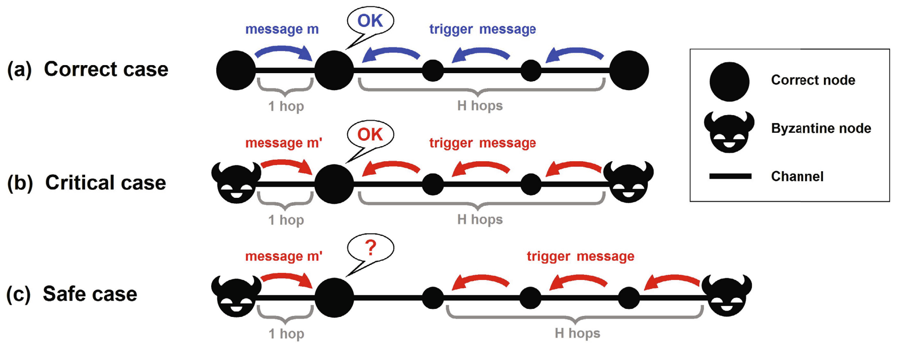


Fig. 1. Illustration of the trigger mechanism

2.2 Notations and Hypotheses

Let (G, E) be a non-oriented graph representing the topology of the network. G denotes the *nodes* of the network. E denotes the *neighborhood* relationship. A node can only send messages to its neighbors. Some nodes are *correct* and follow the protocol described thereafter. We consider that all other nodes are totally unpredictable (or *Byzantine*) and may exhibit an arbitrary behavior.

Hypotheses. We consider an asynchronous message passing network: any message sent is eventually received, but it can be at any time. We assume that, in an infinite execution, any process is activated inifinitely often. However, we make no hypothesis on the order of activation of the processes. Finally, we assume local topology knowledge: when a node receives a message from a neighbor p, it knows that p is the author of the message. Therefore, a Byzantine node cannot lie about its identity to its direct neighbors. This model is referred to as the "oral" model in the literature (or *authenticated channels*).

Messages Formalism. In the protocol, two types of messages can be exchanged:

– *Standard messages*, of the form (m): a message claiming that the source broadcasted the information m.
– *Trigger messages*, of the form (m, S): a message claiming that a node has delivered m. The set S should contain the identifiers of the nodes visited by this message.

The protocol is characterized by a parameter $H \geq 1$: the maximal number of hops that a trigger message can cross. Typically, this limit is reached when S contains more than $H - 1$ nodes. This parameter is known by all correct nodes.

Local Memories. Each correct node p maintains two dynamic sets, initially empty:

– $Wait$: the set of standard messages received, but not yet accepted. When $(m, q) \in Wait$, it means that p received a standard message (m) from a neighbor q.
– $Trig$: set of trigger messages received. When $(m, S) \in Trig$, it means that p received a trigger message $(m, S - \{q\})$ from a neighbor q.

Vocabulary. We will say that a node *multicasts* a message when it sends it to all its neighbors. A node *delivers* a message m when its consider that it is the authentic information broadcast by the source. In the remaining of the paper, we call D the shortest number of hops between two Byzantine nodes. For instance, $D = 4$ in Figure 1-b, and $D = 5$ in Figure 1-c.

2.3 Local Execution of the Protocol

Initially, the source multicasts m and $(m, \emptyset)$. Then, each correct node follows these three rules:

– RECEPTION – When a standard message (m) is received from a neighbor q: if q is the source, deliver m, then multicast (m) and $(m, \emptyset)$; else, add (m, q) to the set $Wait$.
– TRANSMISSION – When a trigger message (m, S) is received from a neighbor q: if $q \notin S$ and $card(S) \leq H - 1$, add $(m, S \cup \{q\})$ to the set $Trig$ and multicast $(m, S \cup \{q\})$.
– DECISION – When there exists (m, q, S) such that $(m, q) \in Wait$, $(m, S) \in Trig$ and $q \notin S$: deliver m, then multicast (m) and $(m, \emptyset)$.

3 Protocol Properties

In this section, we give conditions about the placement of Byzantine nodes that guarantee network *safety* (that is, no correct node ever delivers a false message). Then, we give a methodology to compute a set of nodes that always delivers authentic messages, in any possible execution. Remind that correct nodes do *not* know the actual positions of Byzantine nodes.

3.1 Network Safety

The following theorem guarantees network safety, provided that Byzantine node are sufficiently spaced. This condition depends on the parameter H of the protocol, and on the distance D (see 2.2). We also show that the condition on D is tight for our protocol.

Notice that safety does not guarantee that correct nodes actually *deliver* the authentic message. This aspect is studied in 3.2.

Theorem 1 (Network Safety). *If $D \geq H+2$, no correct nodes delivers a false message.*

Proof. The proof is by contradiction. Let us suppose the opposite : $D \geq H+2$, and at least one correct node delivers a false message. Let u be the first correct node to deliver a false message, and let m' be this message.

No correct node can deliver m' in RECEPTION, as the source did not send m'. So u delivered m' in DECISION, implying that there exists q and S such that $(m', q) \in u.Wait$, $(m', S) \in u.Trig$ and $q \notin S$.

The statement $(m', q) \in u.Wait$ implies that u received (m') from a neighbor q in RECEPTION. Let us suppose that q is correct. Then, q sent (m') in DECISION, implying that q delivered m'. This is impossible, as u is the first correct node to deliver m'. So q is necessarily Byzantine.

Now, let us prove the following property $\mathcal{P}_i$ by recursion, for $1 \leq i \leq H+1$: a correct node u_i, at i hops or less from q, received a message (m', S_i), and $card(S) = card(S_i) + i$.

- First, let us show that $\mathcal{P}_1$ is true. The statement $(m', S) \in u.Trig$ implies that u received $(m', \mathcal{X})$ from a neighbor x in TRANSMISSION, with $S = \mathcal{X} \cup \{x\}$ and $x \notin \mathcal{X}$, So $card(S) = card(\mathcal{X}) + 1$. Therefore, $\mathcal{P}_1$ is true if we take $u_1 = u$ and $S_1 = \mathcal{X}$. Besides, it is also necessary that $card(\mathcal{X}) \leq H-1$, so $card(S) \leq H$.
- Let us suppose that $\mathcal{P}_i$ is true, with $i \leq H$. The node u_i received (m', S_i) from a node x, so x is at $i+1$ hops or less from q. Let us suppose that x is Byzantine. Then, according to the previous statement, $D \leq i+1 \leq H+1$, contradicting our hypothesis. So x is necessarily correct.
  Node x could not have sent (m', S_i) in RECEPTION or DECISION, as u is the first correct node to deliver m'. So this happened in TRANSMISSION, implying that x received $(m', \mathcal{Y})$ from a node y, with $S_i = \mathcal{Y} \cup \{y\}$ and $y \notin \mathcal{Y}$. So $card(S_i) = card(\mathcal{Y}) + 1$, and $card(S) = card(\mathcal{Y}) + i + 1$. Therefore, $\mathcal{P}_{i+1}$ is true if we take $u_{i+1} = x$ and $S_{i+1} = \mathcal{Y}$.

Overall, $\mathcal{P}_{H+1}$ is true and $card(S) = card(S_{H+1}) + H + 1 \geq H + 1$. But, according to a previous statement, $card(S) \leq H$. This contradiction completes the proof.

As a complementary result, let us show that the bound $D \geq H + 2$ is tight for our protocol.

Theorem 2 (Tight bounds for safety). *If $D = H + 1$, some correct nodes may deliver a false message.*

Proof. Let b and c be two Byzantine nodes distant from $H + 1$ hops. Let $(p_0, ..., p_{H+1})$ be a path of $H + 1$ hops, with $p_0 = b$ and $p_{H+1} = c$. Then, b can send a standard message (m') to p_1, and c can send the trigger message for m' trough H hops. Therefore, it is possible that p_1 delivers the false message, and the network is not safe.

3.2 Network Reliability

Here, we suppose that the safety conditions determined in Section 3.1 are satisfied: no correct node can deliver a false message. We now give a methodology to construct a set S of nodes that always delivers the authentic message.

Definition 1 (Reliable node set). *For a given source node and a given distribution of Byzantine nodes, a set of correct nodes S is* reliable *if all nodes in S eventually deliver authentic messages in any possible execution.*

Definition 2 (Correct path). *A N-hops* correct path *is a sequence of distinct correct nodes $(p_0, \ldots, p_N)$ such that, $\forall i \leq N - 1$, p_i and p_{i+1} are neighbors.*

Notice that, according to RECEPTION (see 2.3), the set formed by the source and its correct neighbors is reliable. The following theorem permits to decide whether a given node p can be added to a reliable set S. So, a reliable set can be extended node by node, and can potentially contain the majority or the totality of the correct nodes.

Theorem 3 (Reliable set determination). *Let us suppose that the hypotheses of Theorem 1 (Network Safety) are all satisfied. Let S be a reliable node set, and $p \notin S$ a node with a neighbor $q \in S$. If there exists a correct path of H hops or less between p and a node $v \in S$ (all nodes of the path being distinct from q), then $S \cup \{p\}$ is also a reliable node set.*

Proof. Let m be the message broadcast by the source. As the hypotheses of Theorem 1 are satisfied, the correct nodes can only deliver m. As q and v are in a reliable node set, there exists a configuration where q and v have delivered m. This implies that q and v have multicast (m) and $(m, \emptyset)$.

So p eventually receives (m) from q. If q is the source, p delivers m, completing the proof. Now, let us suppose that q is not the source. Then, p eventually adds (m, q) to its set $Wait$ in RECEPTION.

Let $(v_0, \ldots, v_N)$ be a N-hops correct path, with $v_0 = v$, $v_N = p$ and $N \leq H$. Let S_i be the set of nodes defined by $S_0 = \emptyset$ and $S_i = \{v_0, \ldots, v_{i-1}\}$ for $1 \leq i \leq N$. Let us prove the following property $\mathcal{P}_i$ by induction, for $0 \leq i \leq N - 1$: Node v_i eventually multicasts (m, S_i).

– $\mathcal{P}_0$ is true, as $v_0 = v$ has multicast $(m, \emptyset)$.
– Let us suppose that $\mathcal{P}_i$ is true, with $i \leq N-2$. Let e be an execution where v_i has multicast (m, S_i). Then, v_{i+1} eventually receives (m, S_i). According to TRANSMISSION, as $card(S_i) \leq H-1$ and $v_i \notin S_i$, v_{i+1} eventually multicast (m, S_{i+1}). Therefore, $\mathcal{P}_{i+1}$ is true.

So $\mathcal{P}_{N-1}$ is true and v_{N-1} eventually multicasts (m, S_{N-1}). Therefore, p eventually receives (m, S_{N-1}). According to TRANSMISSION, as $card(S_{N-1}) \leq H-1$ and $v_{N-1} \notin S_{N-1}$, (m, S_{N-1}) is eventually added to $p.Trig$. Thus, we eventually have $(m, q) \in p.Wait$, $(m, S_{N-1}) \in p.Trig$ and $q \notin S_{N-1}$. So according to DECISION, p eventually delivers m.

4 A Reliable Torus Network

In this section, we refined the general conditions given in section 3 for the particular case of torus networks. Torus is good example of a multihop sparse topology, as every node has exactly four neighbors, and is sufficiently regular to permit analytical reasoning.

4.1 Preliminaries

We first recall the definition of the torus topology:

Definition 3 (Torus network). *A $N \times N$* torus *network is a network such that:*

– *Each node has a unique identifier (i, j) with $1 \leq i \leq N$ and $1 \leq j \leq N$.*
– *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 $|j_1 - j_2| = 1$ or N.*
  • *$j_1 = j_2$ and $|i_1 - i_2| = 1$ or N.*

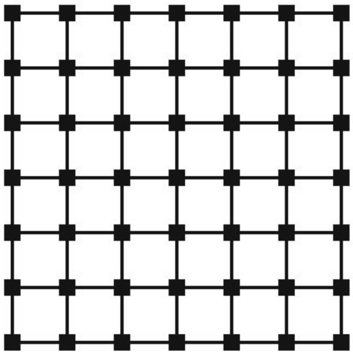


Fig. 2. Example of grid network: a 7×7 grid

Tori vs *Grids.* If we remove the *"or N"* from the previous definition, we obtain an arguably more realistic topology: the *grid.* A grid network can easily be represented in a bidimensional space (see Figure 2).

However, no general condition on the distance between Byzantine nodes can guarantee reliable broadcast in the *grid.* Indeed, let us suppose that the node $(2, 2)$ is the source, and that the node $(1, 2)$ is Byzantine. Then, the node $(1, 1)$ has no way to know which node tells the truth between $(1, 2)$ and $(2, 1)$.

To avoid such border effects, we consider a *torus* network in this part. The grid will be studied in Section 5, with an experimental probabilistic study.

4.2 A Sufficient Condition for Reliable Broadcast

The main theorem of this section guarantees network safety, again in terms of spacing Byzantine nodes apart. This condition depends on the parameter H of the protocol, and on the distance D (see 2.2). We also show that the condition on D is tight for our protocol.

Theorem 4 (Torus reliable broadcast). *Let T be a torus network, and let the parameter of the protocol be $H = 2$. If $D \geq 5$, all correct nodes eventually deliver the authentic message.*

Proof. According to Theorem 1, as $H = 2$ and $D \geq 5$, no correct node ever delivers a false message. In the sequel, the expression *proof by exhaustion* designates a large number of trivial proofs that we do not detail, as they present no particular interest.

If the dimensions of the torus are 5×5 or less, the proof of reliable broadcast is by exhaustion: we consider each possible distribution of Byzantine nodes, and use Theorem 3 to show that all correct nodes eventually deliver the authentic message. Now, let use suppose that the dimensions of the torus are greater than 5×5.

Let v be any correct node. Let $(u_1, \ldots, u_n)$ be a path between the source s and v. If this path is not correct, we can easily construct a correct path between s and v. Indeed, as $D \geq 5$, there exists a square correct path of 8 hops around each Byzantine node. So, for each Byzantine node u_i from the path, we replace u_i by the correct path linking u_{i-1} and u_{i+1}. Therefore, we can always construct a correct path $(p_1, \ldots, p_n)$ between s and v.

For a given node p, we call $G_{3\times3}(p)$ the 3×3 grid from which p is the central node $(2, 2)$, and $G_{5\times5}(p)$ the 5×5 grid from which p is the central node $(3, 3)$. We want to prove the following property $\mathcal{P}_i$ by induction: all correct nodes of $G_{3\times3}(p_i)$ eventually deliver the authentic message.

- We prove $\mathcal{P}_1$ by exhaustion: we consider each possible distribution of Byzantine nodes in $G_{3\times3}(s)$ with $D \geq 5$, and use Theorem 3 to show that all correct nodes eventually deliver the authentic message.
- Let us suppose that $\mathcal{P}_i$ is true. $G_{3\times3}(p_{i+1})$ contains p_i and at least two of its neighbors. As $D \geq 5$, at least one on these neighbors q is correct. As p_i and q are also in $G_{3\times3}(p_i)$, they eventually deliver the authentic message, according to $\mathcal{P}_i$.

- Let us suppose that there is no Byzantine node in $G_{3\times 3}(p_{i+1})$. Then, we prove $\mathcal{P}_{i+1}$ by exhaustion: we consider each possible distribution of Byzantine nodes in $G_{3\times 3}(p_{i+1})$ with $D \geq 5$, and use Theorem 3 to show that all correct nodes eventually deliver the authentic message.
- Let us suppose that there are some Byzantine node in $G_{3\times 3}(p_{i+1})$. According to our hypothesis, there is at most one Byzantine node b in $G_{3\times 3}(p_{i+1})$. Then, all correct nodes of $G_{3\times 3}(p_{i+1})$ are in $G_{5\times 5}(b)$ – so, in particular, p_i and q. As $D \geq 5$, b is the only Byzantine node in $G_{5\times 5}(b)$. Then, we prove $\mathcal{P}_{i+1}$ by exhaustion: we consider each possible placement of p_i and q in $G_{5\times 5}(b)$, and use Theorem 3 to show that all correct nodes of $G_{5\times 5}(b)$ – and thus, all correct nodes of $G_{3\times 3}(p_{i+1})$ – eventually deliver the authentic message.

So $\mathcal{P}_n$ is true, and $v = p_n$ eventually delivers the authentic message.

As a complementary result, let us show that the bound $D \geq 5$ is tight for our protocol.

Theorem 5 (Torus tight bounds). *If $D = 4$, some correct nodes may never deliver the authentic message.*

Proof. Let T be a $N \times N$ torus network, with $N \geq 8$. Let us consider the example given in Figure 3, where $D = 4$. In this figure, the central node s is the source node. As they are direct neighbors of the source, the node of type 1 eventually deliver the authentic message. However, the nodes of type 2 never do so.

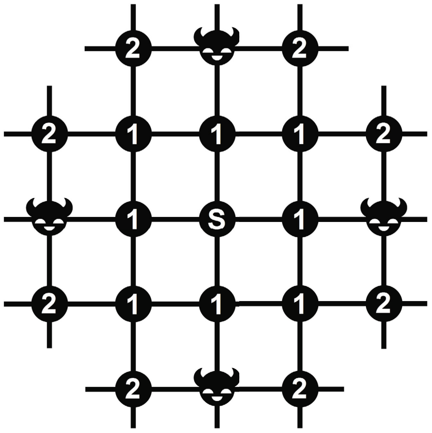


Fig. 3. Critical case in a torus network

Indeed, let us consider a node p of type 2, and its neighbor q of type 1. To deliver the authentic message, p needs to receive a trigger message from another node of type 1, by a correct path of H hops that does not contain q. But, as $H = 2$, such a path does not exist. Besides, we cannot take $H > 2$, as it would enable some correct nodes to deliver a false message, according to Theorem 2. Therefore, the nodes of type 2 – and thus, the other correct nodes – will never deliver the authentic message.

Finally, let us discuss possible extensions to a grid-shaped network. We have seen that perfect reliable broadcast was impossible in a grid, due to border effects. However, it is actually possible in a sub-grid extracted from the grid.

More precisely, let $\mathcal{G}$ be a $N \times N$ grid, and $\mathcal{G}'$ a sub-grid containing all the nodes (i, j) of $\mathcal{G}$ such that $4 \leq i \leq N-4$ and $4 \leq j \leq N-4$. Then, the proof of Theorem 4 is also valid for $\mathcal{G}'$.

It is also the case if we consider *any* particular node in an infinite grid (but not *all* nodes). In other words, a given correct node eventually delivers the authentic message, even if the notion of perfect reliable broadcast does not make sense in an infinite network.

5 Experimental Evaluation

In this section, we target quantitative Byzantine resilience evaluation when considering the case of randomly distributed Byzantine failures. We first give a methodology to estimate the number of Byzantine failures that a particular network can tolerate for a given *probabilistic* guarantee. Then, we present experimental results for a *grid* topology.

Notice that only the *placement* of Byzantine failures is probabilistic: once this placement is determined, we must assume that the Byzantine nodes adopt the worst possible strategy, and that the worst possible execution may occur.

5.1 Methodology

Let n_B be the number of Byzantine failures, randomly distributed on the network (the distribution is supposed to be uniform). We would like to evaluate the *probability* $P(n_B)$, for a correct node, to deliver the authentic message. For this purpose, we use a Monte-carlo method:

- We generate several random distributions of n_B Byzantine failures.
- For each distribution, we randomly choose a source node s and a correct node v. Then, we use Theorem 3 to construct a *reliable node set* (see Definition 1). If v is in the reliable node set, it eventually delivers the authentic message, and the simulation is a success – else, it is a failure.
- With a large number of simulations, the fraction of successes will approximate $P(n_B)$.

More precisely, we approximate a *lower bound* of $P(n_B)$, as the reliable node set constructed in not necessarily the best. Therefore, we can determine a maximal number of Byzantine failures that can be tolerated for a given guarantee (for instance: $P(n_B) \geq 0.99$).

5.2 Results

We run simulations on $N \times N$ grid networks, with a parameter $H = 2$ for the protocol. The results are presented in Figure 4.

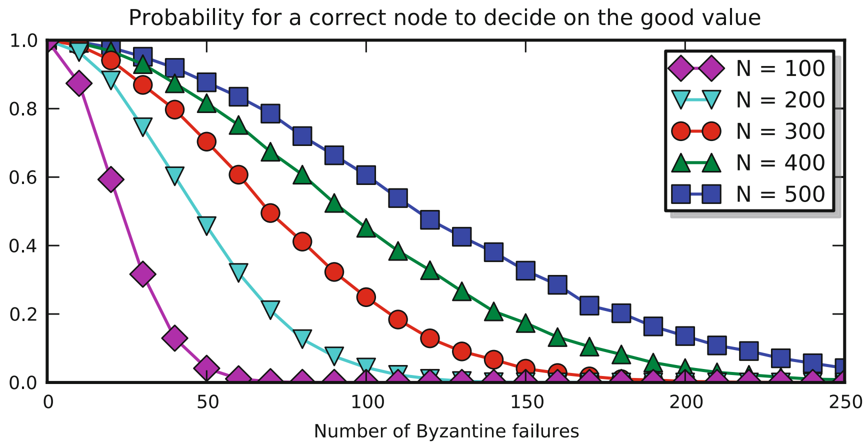


Fig. 4. Experimental evaluations on $N \times N$ grid networks

As expected, a larger grid can tolerate more Byzantine failures, as they are more likely to be sufficiently spaced.

To our knowledge, the only existing protocol working on such a sparse topology – without specific initialization of the nodes – is Explorer [19]. This protocol consists in a voting system on *node-disjoint* paths between the source and the peers. However, as a node has at most 4 neighbors, 2 Byzantine failures can prevent *any* correct node to deliver the authentic message. Therefore, no guarantee can be given for more than 1 Byzantine failure.

As in [16], we could have modified Explorer and forced it to use predetermined paths on the grid. However, this would require global topology knowledge. More precisely, in order to use such a tweaked version of Explorer, a node must know its position on the grid and, for a given neighbor, whether it is its upper, lower, left or right neighbor. Those assumptions are not required with our protocol.

On this grid topology, our protocol enables to tolerate *more* than 1 Byzantine failure with a good probability. For instance, for $N = 500$, we can tolerate up to 14 Byzantine failures with $P(n_B) \geq 0.99$ (see Figure 4).

6 Conclusion

In this paper, we proposed a Byzantine-resilient broadcast protocol for loosely connected networks that does not require any specific initialization of the nodes, nor global topology knowledge. We gave a methodology to construct a reliable node set, then sufficient conditions for perfect reliable broadcast in a sparse topology: the torus. Finally, we presented a methodology to determine the number on randomly distributed Byzantine failures that a network can hold.

Several interesting open questions remain. First, we have the strong intuition that the condition proved on the torus could be generalized to *any* network topology. Another challenging problem is to obtain theoretical probabilistic guarantees, based on global network parameters such as diameter, node degree or connectivity. Third, the tradeoff between global knowledge and the number of Byzantine nodes that can be tolerated requires further attention.

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. In: OSDI, pp. 173–186 (1999)
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. In: 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), pp. 90–98. IEEE, Washington (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)

RMR-Efficient Randomized Abortable Mutual Exclusion[⋆]

(Extended Abstract)

Abhijeet Pareek and Philipp Woelfel

University of Calgary, Canada
{apareek,woelfel}@ucalgary.ca

Abstract. Recent research on mutual exclusion for shared-memory systems has focused on *local spin* algorithms. Performance is measured using the *remote memory references* (RMRs) metric. As common in recent literature, we consider a standard asynchronous shared memory model with N processes, which allows atomic read, write and compare-and-swap (short: CAS) operations.

In such a model, the asymptotically tight upper and lower bounds on the number of RMRs per passage through the Critical Section is $\Theta(\log N)$ for the optimal deterministic algorithms [6,22]. Recently, several *randomized* algorithms have been devised that break the $\Omega(\log N)$ barrier and need only $o(\log N)$ RMRs per passage in expectation [7,13,14]. In this paper we present the first randomized *abortable* mutual exclusion algorithm that achieves a sub-logarithmic expected RMR complexity. More precisely, against a weak adversary (which can make scheduling decisions based on the entire past history, but not the latest coin-flips of each process) every process needs an expected number of $O(\log N/ \log\log N)$ RMRs to enter end exit the critical section. If a process receives an abort-signal, it can abort an attempt to enter the critical section within a finite number of its own steps and by incurring $O(\log N/ \log\log N)$ RMRs.

Keywords: Abortable Mutual Exclusion, Remote Memory References, RMRs, Weak Adversary, Randomization, Shared Memory.

1 Introduction

Mutual exclusion, introduced by Dijkstra [9], is a fundamental and well studied problem. A mutual exclusion object (or lock) allows processes to synchronize access to a shared resource. Each process obtains a lock through a *capture protocol* but at any time, at most one process can own the lock. The owner of a lock can execute a release protocol which frees up the lock. The capture protocol and release protocol are often denoted *entry* and *exit section*, and a process that owns the lock is in the *critical section*.

In this paper, we consider the standard *cache-coherent* (CC) shared model with N processes that supports atomic read, write, and compare-and-swap

⋆ Supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by Alberta Innovates Technology Futures (AITF).

(short: CAS) operations. In this model, all shared registers are stored in globally accessible shared memory. In addition, each process has a local cache and a cache protocol ensures coherency. A *Remote Memory Reference* (short: RMR) is a shared memory access of a register that cannot be resolved locally (i.e., a cache miss). Mutual exclusion algorithms require processes to busy-wait, so the traditional step complexity measure, which counts the number of shared memory accesses, is not useful. Recent research [1,2,4,6,8,16–19] on mutual exclusion algorithms therefore focusses on minimizing the number of remote memory references (RMR). The maximum number of RMRs that any process requires (in any execution) to capture and release a lock is called the RMR complexity of the mutual exclusion algorithm.

Algorithms that perform all busy-waiting by repeatedly reading *locally accessible* shared variables, achieve bounded RMR complexity and have practical performance benefits [4]. Such algorithms are termed *local spin* algorithms. For a comprehensive survey see [3]. Yang and Anderson presented the first $\mathcal{O}(\log N)$ RMRs mutual exclusion algorithm [22] using only reads and writes. Anderson and Kim [1] conjectured that this was optimal, and the conjecture was proved by Attiya, Hendler, and Woelfel [6].

Local spin mutual exclusion locks do not meet a critical demand of many systems [21]. Specifically, the locks employed in database systems and in real time systems must support a “timeout” capability which allows a process that waits “too long” to abort its attempt to acquire the lock. Locks that allow a process to abort its attempt to acquire the lock are called abortable locks. Jayanti presented an efficient deterministic abortable lock [16] with worst-case $\mathcal{O}(\log N)$ RMR complexity, which is optimal for deterministic algorithms.

In this paper we present the first randomized abortable mutual exclusion algorithm that achieves a sub-logarithmic RMR complexity. For randomized algorithms, the influence of random choices made by processes on the scheduling is modeled by an *adversary*. Adversaries of varying powers have been defined. The most common ones are the *oblivious*, the *weak*, and the *adaptive* adversary [5]. An *oblivious* adversary makes all scheduling decisions in advance, before any process flips a coin. This model corresponds to a system, where the coin flips made by processes have no influence on the scheduling. A more realistic model is the *weak* adversary, who sees the coin flip of a process not before that process has taken a step following that coin flip. The *adaptive* adversary models the strongest adversary with reasonable powers, and it can see every coin flip as it appears, and can use that knowledge for any future scheduling decisions. Upper bounds by Hendler and Woelfel [14] and matching lower bounds by Giakkoupis and Woelfel [10] show that the expected RMR complexity of randomized mutual exclusion against an adaptive adversary is $\Theta(\log N/\log\log N)$. Recently Bender and Gilbert [7] presented a randomized lock that has amortized $\mathcal{O}(\log^2 \log N)$ expected RMR complexity against the oblivious adversary. Unfortunately, this algorithm is not strictly deadlock-free (processes may deadlock with small probability, so deadlock has to be expected in a long execution). Our randomized abortable mutual exclusion algorithm is deadlock-free, works against the weak adversary and achieves the same epected RMR complexity as the algorithm by Hendler and Woelfel, namely $\mathcal{O}(\log N/\log\log N)$ expected RMR complexity against the weak adversary.

The randomized algorithm we present uses `CAS` objects and read-write registers. Recently in [12], Golab, Higham and Woelfel demonstrated that using linearizable implemented objects in place of atomic objects in randomized algorithms allows the adversary to change the probability distribution of results. Therefore, in order to safely use implemented objects in place of atomic ones in randomized algorithms, it is not enough to simply show that the implemented objects are linearizable. Also in [12], it is proved that there exists no general correctness condition for the weak adversary, and that the weak adversary can gain additional power depending on the linearizable implementation of the object. Therefore, in this paper we assume that `CAS` operations are atomic.

Abortable Mutual Exclusion. We formalize the notion of an abortable lock by specifying two methods, `lock()` and `release()`, that processes can use to capture and release the lock, respectively. The model assumes that a process may receive a signal to abort at any time during its `lock()` call. If that happens, and only then, the process may fail to capture the lock, in which case method `lock()` returns value $\perp$. Otherwise the process captures the lock, and method `lock()` returns a non-$\perp$ value, and the `lock()` call is deemed *successful.* Note that a `lock()` call may succeed even if the process receives a signal to abort during a `lock()` call.

Code executed by a process after a successful `lock()` method call and before a subsequent `release()` invocation is defined to be its Critical Section. If a process executes a successful `lock()` call, then the process's *passage* is defined to be the `lock()` call, and the subsequent Critical Section and `release()` call, in that order. If a process executes an unsuccessful `lock()` call, then it does not execute the Critical Section or a `release()` call, and the process's passage is just the `lock()` call. Code executed by a process outside of any passage is defined to be its Remainder Section.

The *abort-way* consists of the steps taken by a process during a passage that begins when the process receives a signal to abort and ends when the process returns to its Remainder Section. Since it makes little sense to have an abort capability where processes have to wait for other processes, the abort-way is required to be bounded wait-free (i.e., processes execute the abort-way in a bounded number of their own steps). This property is known as *bounded abort.* Other properties are defined as follows. *Mutual Exclusion*: At any time there is at most one process in the Critical Section; *Deadlock Freedom*: If all processes in the system take enough steps, then at least one of them will return from its `lock()` call; *Starvation Freedom*: If all processes in the system take enough steps, then every process will return from its `lock()` call. The abortable mutual exclusion problem is to implement an object that provides methods `lock()` and release() such that it that satisfies mutual exclusion, deadlock freedom, and bounded abort.

Model. We use the asynchronous shared-memory model [15] with N processes which communicate by executing *operations* on *shared objects*. We consider a system that supports atomic read-write registers and CAS() objects. A CAS object O stores a value from some set and supports two atomic operations O.CAS() and O.Read(). Operation O.Read() returns the value stored in O. Operation O.CAS(exp, new) takes two arguments exp and new and attempts to change the value of O from exp to new. If the value of O equals exp then the operation O.CAS(exp, new) *succeeds*, and the value of O is changed from exp to new, and **true** is returned. Otherwise, the operation fails, and the value of O remains unchanged and **false** is returned.

In addition, a process can execute local coin flip operations that returns an integer value distributed uniformly at random from an arbitrary finite set of integers. The scheduling, generated by the *adversary*, can depend on the random values generated by the processes and we assume the weak adversary model (see for example [5]). We consider the *cache-coherent* (CC) model where each processor has a private cache in which it maintains local copies of shared objects that it accesses. The private cache can be accessed for free. The shared memory is considered remote to all processors. A hardware protocol ensures cache consistency. A memory access to a shared object that requires access to remote memory is called a *remote memory reference* (RMR). The *RMR complexity* of a algorithm is the maximum number of RMRs that a process can incur during any execution of the algorithm.

Results. We present several building blocks for our algorithm in Section 1. In Sections 2 and 3 we give an overview of the randomized mutual exclusion algorithm. Due to lack of space full proofs are omitted from this extended abstract. A more precise description of our algorithms including pseudo code, as well as a full analysis can be found in the full version of the paper [20]. Our results are summarized by the following theorem.

Theorem 1. *There exists a starvation-free randomized abortable N process lock against the weak adversary, where a process incurs $\mathcal{O}(\log N/\log\log N)$ RMRs in expectation per passage. The lock requires $\mathcal{O}(N)$* CAS *objects and read-write registers.*

A Randomized CAS Counter. A *CAS counter* object with parameter $k \in \mathbb{Z}_{\geq 0}$ complements a CAS object by supporting an additional inc() operation that increments the object's value. The object takes values in $\{0,\ldots,k\}$, and initially the object's value is 0. Operation inc() takes no arguments, and if the value of the object is in $\{0,\ldots,k-1\}$, then the operation increments the value and returns the previous value. Otherwise, the value of the object is unchanged and the integer k is returned. We will use such an object for $k = 2$ to assign three distinct roles to processes.

Our implementation of the inc() operation needs only $O(1)$ RMRs in expectation. A deterministic implementation of a CAS counter for $k = 2$ and constant worst-case RMR complexity does not exist: Replacing our randomized CAS counter with a deterministic one that has worst-case RMR complexity T yields a deterministic abortable mutual exclusion algorithm with worst-case RMR complexity $\mathcal{O}(T \cdot \log N / \log\log N)$. From the lower bound for deterministic mutual exclusion by Attiya etal. [6], such an algorithm does not exist, unless $T = \Omega(\log\log N)$. (For the DSM model, a super-constant lower bound on the RMR complexity of a CAS counter actually follows from an earlier result by Golab, Hadzilacos, Hendler, and Woelfel [11].)

Our randomized CAS counter, which we call $\mathsf{RCAScounter}_k$, allows the `inc()` method to fail. The idea is, that to increase the value of the object, a process randomly guesses its current value, v, and then executes a `CAS(`v,v+1`)` operation. An adaptive adversary could intervene between the steps involving the random guess and the subsequent CAS operation, thereby affecting the failure probability of an `inc()` method call, but a weak adversary cannot do so.

Lemma 1. *Object* $\mathsf{RCAScounter}_k$ *is a randomized wait-free linearizable CAS Counter, where the probability that an* `inc()` *method call fails is* $\frac{k}{k+1}$ *against the weak adversary, and each method has* $\mathcal{O}(1)$ *(worst-case) step complexity.*

The Single-Fast-Multi-Slow Universal Construction. A *universal construction* object provides a linearizable concurrent implementation of any object with a sequential specification that can be given by deterministic code. We devise a universal construction object $\mathsf{SFMSUnivConst}\langle\mathsf{T}\rangle$ for N processes which provides two methods, `doFast(`*op*`)` and `doSlow(`*op*`)`, to perform any operation *op* on an object of type T. The idea is that `doFast()` methods cannot be called concurrently, but are executed very fast, i.e., they have $\mathcal{O}(1)$ step complexity. On the other hand, `doSlow()` methods may need up to $\mathcal{O}(N)$ steps. Later, we use the universal construction object for smaller sets of $\Delta = \mathcal{O}(\log N / \log\log N)$ processes, and then the step complexity of `doSlow()` is bounded by $\mathcal{O}(\Delta)$. The algorithm is based on a helping mechanism in which `doSlow()` methods help a process that wants to execute a `doFast()` method.

Lemma 2. *Object* $\mathsf{SFMSUnivConst}\langle\mathsf{T}\rangle$ *is a wait-free universal construction that implements an object* $\mathbb{O}$ *of type* T*, for* N *processes, and an operation op on object* $\mathbb{O}$ *is performed by executing either method* `doFast(op)` *or* `doSlow(op)`*, and no two processes execute method* `doFast()` *concurrently. Methods* `doFast()` *and* `doSlow()` *have* $\mathcal{O}(1)$ *and* $\mathcal{O}(N)$ *step complexity respectively.*

The Abortable Promotion Array. An object O of type $\mathsf{AbortableProArray}_k$ stores a vector of k integer pairs. It provides some specialized operations on the vector, such as conditionally adding/removing elements, and earmarking a process (associated with an element of the vector) for some future activity. Initially the value of $O = (O[0], O[1], \ldots, O[k-1])$ is $(\langle 0, \bot\rangle, \ldots, \langle 0, \bot\rangle)$. The object supports operations `collect()`, `abort()`, `promote()`, `remove()` and `reset()`. Operation `collect(`X`)` takes as argument an array $X[0 \ldots k-1]$ of integers, and is used to "register" processes into the array. The operation changes $O[i]$, for all

i in $\{0,\ldots,k-1\}$, to value $\langle \mathsf{REG}, X[i]\rangle$ except if $O[i]$ is $\langle \mathsf{ABORT}, s\rangle$, for some $s \in \mathbb{Z}$. In the latter case the value of $O[i]$ is unchanged. Process i is said to be *registered* in the array if a `collect()` operation changes $O[i]$ to value $\langle \mathsf{REG}, s\rangle$, for some $s \in \mathbb{Z}$. The object also allows processes to "abort" themselves from the array using the operation `abort()`. Operation `abort(`i, s`)` takes as argument the integers i and s, where $i \in \{0,\ldots,k-1\}$ and $s \in \mathbb{Z}$. The operation changes $O[i]$ to value $\langle \mathsf{ABORT}, s\rangle$ and returns **true**, only if $O[i]$ is not equal to $\langle \mathsf{PRO}, s'\rangle$, for some $s' \in \mathbb{Z}$. Otherwise the operation returns **false**. Process i *aborts* from the array if it executes an `abort(`i, s`)` operation that returns **true**. A registered process in the array that has not aborted can be "promoted" using the `promote()` operation. Operation `promote()` takes no arguments, and changes the value of the element in O with the smallest index and that has value $\langle \mathsf{REG}, s\rangle$, for some $s \in \mathbb{Z}$, to value $\langle \mathsf{PRO}, s\rangle$, and returns $\langle i, s\rangle$, where i is the index of that element. If there exists no element in O with value $\langle \mathsf{REG}, s\rangle$, for some $s \in \mathbb{Z}$, then O is unchanged and the value $\langle \perp, \perp\rangle$ is returned. Process i is *promoted* if a `promote()` operation returns $\langle i, s\rangle$, for some $s \in \mathbb{Z}$. Operation `reset()` resets the entire array to its initial state.

Note that an aborted process in the array, cannot be registered into the array or be promoted, until the array is reset. If a process tries to abort itself from the array but finds that it has already been promoted, then the abort fails. This ensures that a promoted process takes responsibility for some activity that other processes expect of it.

In our abortable lock, the i-th element of the array stores the current state of process with ID i, and a sequence number associated with the state. Operation `collect()` is used to register a set of participating processes into the array. Operation `abort(`i, s`)` is executed only by process i, to abort from the array. Operation `promote()` is used to promote an unaborted registered process from the array, so that the promoted process can fulfill some future obligation.

In our abortable lock of Section 2, we need a wait-free linearizable implementation of type $\mathsf{AbortableProArray}_{\Delta}$, where Δ is the maximum number of processes that can access the object concurrently, and we achieve this by using object $\mathsf{SFM\text{-}SUnivConst}\langle \mathsf{AbortableProArray}_{\Delta}\rangle$. We ensure that no two processes execute operations `collect()`, `promote()`, `reset()` or `remove()` concurrently, and therefore by we get $\mathcal{O}(1)$ step complexity for these operations by using method `doFast()`. Operation `abort()` has $\mathcal{O}(\Delta)$ step complexity since it is performed using method `doSlow()`, which allows processes to call `abort()` concurrently.

2 The Tree Based Abortable Lock

As in the algorithm by Hendler and Woelfel [14], we consider a tree with N leafs and where each non-leaf node has Δ children. Every non-leaf node is associated with a lock. Each process is assigned a unique leaf in the tree and climbs up the tree by capturing the locks on nodes on its path until it has captured the lock at the root. Once a process locks the root, it can enter the Critical Section.

The main difficulty is that of designing the locks associated with the nodes of the tree. A simple `CAS` object together with an "announce array" as used in [14] does not work. Suppose a process p captures locks of several nodes on its path up to the root and aborts before capturing the root lock. Then it must release all captured node locks and therefore these lock releases cause other processes, which are busy-waiting on these nodes, to incur RMRs. So we need a mechanism to guarantee some progress to these processes, while we also need a mechanism that allows busy-waiting processes to abort their attempts to capture node locks. In [14] progress is achieved as follows: A process p, before releasing a lock on its path, searches(with a random procedure) for other processes that are busy-waiting for the node lock to become free. If p finds such a process, it promotes it into the critical section. This is possible, because at the time of the promotion p owns the root lock and can hand it over to a promoted process. Unfortunately, this promotion mechanism fails for abortable mutual exclusion: When p aborts its own attempt to enter the Critical Section, it may have to release node locks at a time when it doesn't own the root lock. Another problem is that if p finds a process q that is waiting for p to release a node-lock, then q may have already decided to abort. We use a carefully designed synchronization mechanism to deal with such cases.

To ensure that waiting processes make some progress, we desire that p "collect" busy-waiting processes (if any) at a node into an instance of an object of type $\mathsf{AbortableProArray}_{\Delta}$, $\mathsf{PawnSet}$, using the operation `collect()`. Once busy-waiting processes are collected into $\mathsf{PawnSet}$, p can identify a busy-waiting process, if present, using the $\mathsf{PawnSet}$.`promote()` operation, while busy-waiting processes themselves can abort using the $\mathsf{PawnSet}$.`abort()` operation. Note that p may have to read $\mathcal{O}(\Delta)$ registers just to find a single busy-waiting process at a node, where Δ is the branching factor of the arbitration tree. This is problematic since our goal is to bound the number of steps during a passage to $\mathcal{O}(\Delta)$ steps, and thus a process cannot collect at more than one node. For this reason we desire that p transfer all unreleased node locks that it owns to the first busy-waiting process it can find, and then it would be done. And if there are no busy-waiting processes at a node, then p should somehow be able to release the node lock in $\mathcal{O}(1)$ steps. Since there are at most Δ nodes on a path to the root node, p can continue to release captured node locks where there are no busy-waiting processes, and thus not incur more than $\mathcal{O}(\Delta)$ overall. We use an instance of $\mathsf{RCAScounter}_2$, Ctr, to help decide if there are any busy-waiting processes at a node lock. Initially, Ctr is 0, and processes attempt to increase Ctr using the Ctr.`inc()` operation after having registered at the node. Process p attempts to release a node lock by first executing a $\mathsf{Ctr}.\mathtt{CAS}(1,0)$ operation. If the operation fails then some process q must have further increased Ctr from 1 to 2, and thus p can transfer all unreleased locks to q, if q has not aborted itself. If q has aborted, then q can perform the collect at the node lock for p, since q can afford to incur an additional one-time expense of $\mathcal{O}(\Delta)$ RMRs. If q has not aborted then p can transfer its captured locks to q in $\mathcal{O}(1)$ steps, and thus making sure some process makes progress towards capturing the root lock. We encapsulate these mechanisms in a randomized abortable lock object, $\mathsf{ALockArray}_{\Delta}$.

More generally, we specify an object $\mathsf{ALockArray}_n$ for an arbitrary parameter $n < N$. Object $\mathsf{ALockArray}_n$ provides methods `lock()` and `release()` that can be accessed by at most $n+1$ processes concurrently. The object is an abortable lock, but with an RMR complexity of $O(n)$ for the abort-way, and constant RMR complexity for `lock()`. The `release()` method is special. If it detects contention (i.e., other processes are busy-waiting), then it takes $O(n)$ RMRs, but helps those other processes to make progress. Otherwise, it takes only $O(1)$ RMRs. Each non-leaf node u in our abritration tree will be associated with a lock $\mathsf{ALockArray}_\Delta$ and can only be accessed concurrently by the processes owning locks associated with the children of u and one other process.

Method `lock()` takes a single argument, which we will call pseudo-ID, with value in $\{0, \ldots, n-1\}$. We denote a `lock()` method call with argument i as $\texttt{lock}_i\texttt{()}$, but refer to $\texttt{lock}_i\texttt{()}$ as `lock()` whenever the context of the discussion is not concerned with the value of i. Method `lock()` returns a non-$\perp$ value if a process captures the lock, otherwise it returns a $\perp$ value to indicate a failed `lock()` call. A `lock()` by process p can fail only if p aborts during the method call. Method `release()` takes two arguments, a pseudo-ID $i \in \{0, \ldots, n-1\}$ and an integer j. Method $\texttt{release}_i\texttt{(}j\texttt{)}$ returns **true** if and only if there exists a concurrent call to `lock()` that eventually returns j. Otherwise method $\texttt{release}_i\texttt{(}j\texttt{)}$ returns **false**. The information contained in argument j determines the transfered node locks. Process pseudo-IDs are passed as arguments to the methods to allow the ability for a process to "transfer" the responsibility of releasing the lock to another process. Specifically, we desire that if a process p executes a successful $\texttt{lock}_i\texttt{()}$ call and becomes the owner of the lock, then p does not have to release the lock itself, if it can find some process q to call $\texttt{release}_i\texttt{()}$ on its behalf. In Section 3 we implement object $\mathsf{ALockArray}_n$ with the properties described the following lemma. (The proof can be found in the full version of this paper [20].)

Lemma 3. *Object* $\mathsf{ALockArray}_n$ *can be implemented against the weak adversary for the CC model with the following properties using only* $\mathcal{O}(n)$ `CAS` *objects and read-write registers.*

(a) Mutual exclusion, starvation freedom, bounded exit, and bounded abort.
(b) The abort-way has $\mathcal{O}(n)$ *RMR complexity.*
(c) If a process does not abort during a `lock()` *call, then it incurs* $\mathcal{O}(1)$ *RMRs in expectation during the call, otherwise it incurs* $\mathcal{O}(n)$ *RMRs in expectation during the call.*
(d) If a process' call to `release(`j`)` *returns* **false***, then it incurs* $\mathcal{O}(1)$ *RMRs during the call, otherwise it incurs* $\mathcal{O}(n)$ *RMRs during the call.*

High Level Description. We use a complete Δ-ary tree $\mathcal{T}$ of depth Δ with N leaves, called the arbitration tree. The height of node u is denoted h_u (leafs have height 0). Each process p is associated with a unique leaf leaf_p in the tree, and path_p denotes the path from leaf_p up to the root, called root.

Each node of our arbitration tree $\mathcal{T}$ is a structure of type Node that contains a single instance L of the abortable randomized lock object $\mathsf{ALockArray}_\Delta$. This object enables processes to abort their attempt at any point during their ascent to the root node. During $\texttt{lock}_p\texttt{()}$ a process p attempts to *capture* every node on its path path_p that it does not own, as long as p has not received a signal to abort. Process p attempts to capture a node u by executing a call to $u.\mathsf{L}.\texttt{lock()}$. If p's $u.\mathsf{L}.\texttt{lock()}$ call returns ∞ then p is said to have captured u, and if the call returns an integer j, then p is said to have been *handed over* all nodes from u to v on path_p, where $\mathsf{h}_v = j$. We ensure that $j \geq \mathsf{h}_u$. Process p starts to *own* node u when p captures $u.\mathsf{L}$ or when p is handed over node u from the previous owner of node u. Process p can enter its Critical Section when it owns the root node of $\mathcal{T}$. Process p may receive a signal to abort during a call to $u.\mathsf{L}.\texttt{lock()}$ as a result of which p's call to $u.\mathsf{L}.\texttt{lock()}$ returns either $\perp$ or a non-$\perp$ value. In either case, p then calls $\texttt{release}_p\texttt{()}$ to release all locks of nodes that p has captured in its passage, and then returns from its $\texttt{lock}_p\texttt{()}$ call with value $\perp$.

An exiting process p *releases* all nodes that it owns during $\texttt{release}_p\texttt{()}$. Process p is said to *release* node u if p releases $u.\mathsf{L}$ (by executing $u.\mathsf{L}.\texttt{release()}$ call), or if p hands over node u to some other process. Recall that p hands over node u if p executes a $v.\mathsf{L}.\texttt{release}(j)$ call that returns **true** where $\mathsf{h}_v \leq \mathsf{h}_u \leq j$. Let s be the height of the highest node p owns. During $\texttt{release}_p\texttt{()}$, p climbs up $\mathcal{T}$ and calls $u.\mathsf{L}.\texttt{release}_p(s)$ at every node u that it owns, until a call returns **true**. If a $u.\mathsf{L}.\texttt{release}_p(s)$ call returns **false** (process p incurs $\mathcal{O}(1)$ steps), then p is said to have released lock $u.\mathsf{L}$ (and therefore released node u), and thus p continues on its path. If a $u.\mathsf{L}.\texttt{release}_p(s)$ call returns **true** (process p incurs $\mathcal{O}(\Delta)$ steps), then p has handed over all remaining nodes that it owns to some process that is executing a concurrent $u.\mathsf{L}.\texttt{lock()}$ call at node u, and thus p does not release any more nodes.

Notice that our strategy to release node locks is to climb up the tree until all node locks are released or a hand over of remaining locks is made. Climbing up the tree is necessary (as opposed to climbing down) in order to hand over node locks to a process, say q, such that the handed over nodes lie on path_q.

3 The Array Based Abortable Lock

We specified object $\mathsf{ALockArray}_n$ in Section 2 and now we describe and implement it (see Figures 1 and 2). Let L be an instance of object $\mathsf{ALockArray}_n$.

Registering and Roles at Lock L. At the beginning of a $\texttt{lock()}$ call processes *register* themselves in the apply array by swapping the value REG atomically into their designated slots (apply[i] for process with pseudo-ID i) using a $\texttt{CAS}$ operation. The array apply of n $\texttt{CAS}$ objects is used by processes to register and "deregister" themselves from lock L, and to notify each other of certain events at lock L.

On registering in the apply array, processes attempt to increase Ctr, an instance of $\mathsf{RCAScounter}_2$, using operation $\mathsf{Ctr}.\texttt{inc()}$. Recall that $\mathsf{RCAScounter}_2$ is a bounded counter, initially 0, and returns values in $\{0, 1, 2\}$ (see Section 1). Each of these values corresponds to a *role* that a processs can assume at lock L. There are actually four roles, *king*, *queen*, *pawn* and *promoted pawn*, which

Object $\mathsf{ALockArray}_n$

```
shared:
   Ctr: RCAScounter$_2$ init 0;
   PawnSet: Object of type AbortableProArray$_n$ init $\varnothing$;
   apply: array $[0 \ldots n-1]$ of int pairs init all $\langle \bot, \bot \rangle$;
   Role: array $[0 \ldots n-1]$ of int init $\bot$;
   Sync1, Sync2: int init $\bot$;
   KING, QUEEN, PAWN, PAWN_P, REG, PRO: const int $0, 1, 2, 3, 4, 5$
   respectively;
   getSequenceNo(): returns integer $k$ on being called for the $k$-th time from
   a call to lock$_i$(). (Since calls to lock$_i$() are executed sequentially, a
   sequential shared counter suffices to implement method getSequenceNo().)
local:
   $s, val, seq, dummy$: int init $\bot$;
   $flag, r$: boolean init false;
   $A$: array $[0 \ldots n-1]$ of int init $\bot$
   // If process $i$ satisfies the loop condition in line 2, 7,
      or 14, and $i$ has received a signal to abort, then $i$ calls
      abort$_i$()
```

Method lock$_i$( )

```
1  $s \leftarrow$ getSequenceNo()
2  await (apply[$i$].CAS($\langle \bot, \bot \rangle, \langle$REG$, s \rangle$))
3  $flag \leftarrow$ true
4  repeat
5      Role[$i$] $\leftarrow$ Ctr.inc()
6      if (Role[$i$] = PAWN) then
7          await (apply[$i$] =
           $\langle$PRO$, s \rangle \vee$ Ctr.Read() $\neq 2$)
8          if (apply[$i$] = $\langle$PRO$, s \rangle$) then
9              Role[$i$] $\leftarrow$ PAWN_P
10         end
11     end
12 until
   (Role[$i$] $\in$ {KING, QUEEN, PAWN_P})
13 if (Role[$i$] = QUEEN) then
14     await (Sync1 $\neq \bot$)
15 end
16 apply[i].CAS($\langle$REG$, s \rangle, \langle$PRO$, s \rangle$)
17 if Role[$i$] = QUEEN then return
   Sync1 else return $\infty$
```

Method abort$_i$( )

```
18 if $\neg flag$ then return $\bot$
19 apply[i].CAS($\langle$REG$, s \rangle, \langle$PRO$, s \rangle$)

20 if Role[$i$] = PAWN then
21     if $\neg$PawnSet.abort($i, s$)
       then
22         Role[$i$] $\leftarrow$ PAWN_P
23         return $\infty$
24     end
25 else
26     if $\neg$Sync1.CAS($\bot, \infty$)
       then
27         return Sync1
28     end
29     doCollect$_i$()
30     helpRelease$_i$()
31 end
32 apply[$i$].CAS($\langle$PRO$, s \rangle, \langle \bot, \bot \rangle$)

33 return $\bot$
```

Method doCollect$_i$()

```
51 for $k \leftarrow 0$ to $n-1$ do
52     $\langle val, seq \rangle \leftarrow$ apply[$k$]
53     if $val$ = REG then $A[k] \leftarrow seq$ else $A[k] \leftarrow \bot$
54 end
55 PawnSet.collect($A$)
```

Fig. 1. Implementation of Object $\mathsf{ALockArray}_n$

```
Method release_i(int j)
34 $r \leftarrow$ false
35 if Role[$i$] = KING then
36     if ¬Ctr.CAS(1, 0) then
37         $r \leftarrow$ Sync1.CAS(⊥, $j$)
38         if $r$ then
           doCollect_i()
39         helpRelease_i()
40     end
41 end
42 if Role[$i$] = QUEEN then
43     helpRelease_i()
44 end
45 if Role[$i$] = PAWN_P then
46     doPromote_i()
47 end
48 $\langle dummy, s\rangle \leftarrow$ apply[$i$]
49 apply[$i$].CAS($\langle$PRO, $s\rangle$, $\langle\perp,\perp\rangle$)

50 return $r$
```

```
Method helpRelease_i()
56 if ¬Sync2.CAS(⊥, $i$) then
57     $j \leftarrow$ Sync1.Read()
58     Sync1.CAS($j$, ⊥)
59     $j \leftarrow$ Sync2.Read()
60     Sync2.CAS($j$, ⊥)
61     PawnSet.remove($j$)
62     doPromote_i()
63 end
```

```
Method doPromote_i()
64 PawnSet.remove($i$)
65 $\langle j, seq\rangle \leftarrow$ PawnSet.promote()
66 if $j = \perp$ then
67     PawnSet.reset()
68     Ctr.CAS(2, 0)
69 else
70     apply[$j$].CAS($\langle$REG, $seq\rangle$, $\langle$PRO, $seq\rangle$)

71 end
```

Fig. 2. Implementation of Object $\mathsf{ALockArray}_n$ (continued)

define the protocols processes follow. During an execution, Ctr cycles from its initial value 0 to non-0 values and then back to 0, multiple times, and we refer to each such cycle as a Ctr-cycle. A process that increases Ctr from 0 to 1 becomes king. A process that increases Ctr from 1 to 2 becomes queen. All processes that attempt to increase Ctr any further, receive the return value 2 and they become pawns. A pawn busy-waits until it gets "promoted" at lock L (a process is said to be *promoted* at lock L if it is promoted in PawnSet), or until it sees the Ctr value decrease, so that it can attempt to increase Ctr again. The algorithm guarantees that a pawn process repeats an attempt to increase Ctr at most once, before getting promoted. We also ensure that at any point in time during the execution, the number of processes that have assumed the role of a king, queen and promoted pawn at lock L, respectively, is at most one, and thus we refer to them as king_L, $\mathsf{queen}_\mathsf{L}$ and $\mathsf{ppawn}_\mathsf{L}$, respectively. We describe the protocol associated with each of the roles in more detail shortly. An array Role of n read-write registers is used by processes to record their role at lock L.

Busy-Waiting in Lock L. The king process, king_L, becomes the first owner of lock L during the current Ctr-cycle, and can proceed to enter its Critical Section, and thus it does not busy-wait during `lock()`. The queen process, $\mathsf{queen}_\mathsf{L}$, must wait for king_L for a notification of its turn to own lock L. Then $\mathsf{queen}_\mathsf{L}$ spins on CAS object Sync1, waiting for king_L to CAS some integer value into Sync1. Process king_L attempts to CAS an integer j into Sync1 only during its call to release(j), after it has executed its Critical Section. The pawn processes wait on their individual slots of the apply array for a notification of their promotion.

A *Collect* Action at Lock L. A collect action is conducted by either king_L during a call to release(), or by queen_L during a call to abort(). A collect action is defined as the sequence of steps executed by a process during a call to doCollect(). During a call to doCollect(), the collecting process (say q) iterates over the array apply reading every slot, and then creates a local array A from the values read and stores the contents of A in the PawnSet object in using the operation PawnSet.collect(A). A key point to note is that operation PawnSet.collect(A) does not overwrite an aborted process's value in PawnSet (a process aborts itself in PawnSet by executing a successful PawnSet.abort() operation).

A *Promote* Action at Lock L. Operation PawnSet.promote() during a call to method doPromote() is defined as a promote action. The operation returns the pseudo-ID of a process that was collected during a collect action, and has not yet aborted from PawnSet. A promote action is conducted at lock L either by king_L, queen_L or ppawn_L.

Lock *Handover* from king_L to queen_L. As mentioned, process queen_L waits for king_L to finish its Critical Section and then call release(j). During king_L's release(j) call, king_L attempts to swap integer j into CAS object Sync1, that only king_L and queen_L access. If queen_L has not "aborted", then king_L successfully swaps j into Sync1, and this serves as a notification to queen_L that king_L has completed its Critical Section, and that queen_L may now proceed to enter its Critical Section.

***Aborting* an Attempt at Lock L by queen_L.** On receiving a signal to abort, queen_L abandons its lock() call and executes a call to abort() instead. queen_L first changes the value of its slot in the apply array from REG to PRO, to prevent itself from getting collected in future collects. Since king_L and queen_L are the first two processes at L, king_L will eventually try to handover L to queen_L. To prevent king_L from handing over lock L to queen_L, queen_L attempts to swap a special value ∞ into Sync1 in one atomic step. If queen_L fails then this implies that king_L has already handed over L to queen_L, and thus queen_L returns from its call to abort() with the value written to Sync1 by king_L, and becomes the owner of L. If queen_L succeeds then queen_L is said to have successfully aborted, and thus king_L will eventually fail to hand over lock L. Since queen_L has aborted, queen_L now takes on the responsibility of collecting all registered processes in lock L, and storing them into the PawnSet object. After performing a collect, queen_L then synchronizes with king_L again, to perform a promote, where one of the collected processes is promoted. After that, queen_L deregisters from the apply array by resetting its slot to the initial value $\langle \perp, \perp \rangle$.

***Aborting* an Attempt at Lock L by a Pawn Process.** On receiving a signal to abort a pawn process (say p) busy-waiting in lock L, abandons its lock() call and executes a call to abort() instead. Process p first changes the value of its slot in the apply array from REG to PRO, to prevent itself from getting collected in future collects. It then attempts to abort itself in PawnSet by executing the operation PawnSet.`abort`(p)). If p's attempt is unsuccessful then it implies that p has already been promoted in PawnSet, and thus p can assume the role of a promoted pawn, and become the owner of L. In this case, p returns from its `abort()` call with value ∞ and becomes the owner of L. If p's attempt is successful then p cannot be collected or promoted in future collects and promotion events. In this case, p deregisters from the apply array by resetting its slot to the initial value $\langle \perp, \perp \rangle$, and returns $\perp$ from its call to `abort()`.

Releasing Lock L. Releasing lock L can be thought of as a group effort between the $\mathsf{king_L}$, $\mathsf{queen_L}$ (if present at all), and the promoted pawns (if present at all). To completely release lock L, the owner of L needs to reset Ctr back to 0 for the next Ctr-cycle to begin. However, the owner also has an obligation to hand over lock L to the next process waiting in line for lock L. We now discuss the individual strategies of releasing lock L, by $\mathsf{king_L}$, $\mathsf{queen_L}$ and the promoted processes. To release lock L, the owner of L executes a call to `release`(j), for some integer j.

Synchronizing the Release of Lock L by $\mathsf{king_L}$ and $\mathsf{queen_L}$. Process $\mathsf{king_L}$ first attempts to decrease Ctr from 1 to 0 using a CAS operation. If it is successful, then $\mathsf{king_L}$ was able to end the Ctr-cycle before any process could increase Ctr from 1 to 2. Thus, there was no $\mathsf{queen_L}$ process or pawn processes waiting for their turn to own lock L, during that Ctr-cycle. Then $\mathsf{king_L}$ is said to have released lock L.

If $\mathsf{king_L}$'s attempt to decrease Ctr from 1 to 0 fails, then $\mathsf{king_L}$ knows that there exists a $\mathsf{queen_L}$ process that increased Ctr from 1 to 2. Since $\mathsf{queen_L}$ is allowed to abort, releasing lock L is not as straight forward as raising a flag to be read by $\mathsf{queen_L}$. Therefore, $\mathsf{king_L}$ attempts to synchronize with $\mathsf{queen_L}$ by swapping the integer j into the object Sync1 using a Sync1.CAS($\perp, j$) operation. Recall that $\mathsf{queen_L}$ also attempts to swap a special value ∞ into object Sync1 using a Sync1.CAS($\perp, j$) operation, in order to abort its attempt. Clearly only one of them can succeed. If $\mathsf{king_L}$ succeeds, then $\mathsf{king_L}$ is said to have successfully handed over lock L to $\mathsf{queen_L}$. If $\mathsf{king_L}$ fails, then $\mathsf{king_L}$ knows that $\mathsf{queen_L}$ has aborted and thus $\mathsf{king_L}$ then tries to hand over its lock to one of the waiting pawn processes. The procedure to hand over lock L to one of the waiting pawn processes is to execute a collect action followed by a promote action.

The collect action needs to be executed only once during a Ctr-cycle, and thus we let the process (among $\mathsf{king_L}$ or $\mathsf{queen_L}$) that successfully swaps a value into Sync1, execute the collect action.

If $\mathsf{king_L}$ successfully handed over L to $\mathsf{queen_L}$, it collects the waiting pawn processes, so that eventually when $\mathsf{queen_L}$ is ready to release lock L, $\mathsf{queen_L}$ can simply execute a promote action. Since there is no guarantee that $\mathsf{king_L}$ will finish collecting before $\mathsf{queen_L}$ desires to execute a promote action, the processes synchronize among themselves again, to execute the first promote action of the current Ctr-cycle. They both attempt to swap their pseudo-IDs into an empty CAS object Sync2, and therefore only one can succeed. The process that is unsuccessful, is the second among them, and therefore by that point the collection of the waiting pawn process must be complete. Then the process that is unsuccessful, resets Sync1 and Sync2 to their initial value $\perp$, and then executes the promote action, where a waiting pawn process is promoted and handed over lock L. If no process were collected during the Ctr-cycle, or all collected pawn processes have successfully aborted before the promote action, then the promote action fails, and thus the owner process resets the PawnSet object, and then resets Ctr from 2 to 0 in one atomic step, thus releasing lock L, and resetting the Ctr-cycle.

The Release of Lock L by $\mathsf{ppawn}_\mathsf{L}$. If a process was promoted by king_L or $\mathsf{queen}_\mathsf{L}$ as described above, then the promoted process is said to be handed over the ownership of L, and becomes the first promoted pawn of the Ctr-cycle. Since a collect for this Ctr-cycle has already been executed, process $\mathsf{ppawn}_\mathsf{L}$ does not execute any more collects, but simply attempts to hand over lock L to the next collected process by executing a promote action. This sort of promotion and handing over of lock L continues until there are no more collected processes to promote, at which point the last promoted pawn resets the PawnSet object, and then resets Ctr from 2 to 0 in one atomic step, thus releasing lock L, and resetting the Ctr-cycle.

All owner processes also *deregister* themselves from lock L, by resetting their slot in the apply array to the initial value $\langle\perp,\perp\rangle$. This step is the last step of their `release(`j`)` calls, and processes return a boolean to indicate whether they successfully wrote integer j into Sync1 during their `release(`j`)` call. Note that only king_L could possibly return **true** since it is the only process that attempts to do so, during its `release(`j`)` call.

4 Conclusion

We presented the first randomized abortable lock that achieves sub-logartihmic expected RMR complexity. While the speed-up is only a modest $O(\log\log n)$ factor over the most efficient deterministic abortable mutual exclusion algorithm, our result shows that randomization can help in principle, to improve the efficiency of abortable locks. Unfortunately, our algorithm is quite complicated; it would be nice to find a simpler one. It would also be interesting to find an abortable algorithm with sub-logarithmic RMR complexity that works against the adaptive adversary. In the weak adversary model, no non-trivial lower bounds for mutual exclusion are known, but it seems hard to improve upon $O(\log n/\log\log n)$ RMR complexity, even without the abortability property. As shown by Bender and Gilbert, [7], mutual exclusion can be solved much more efficiently in the oblivious adversary model. However, their algorithm is not lock-free with probability one.

Acknowledgement. We are indebted to Lisa Higham and Bill Sands for their careful reading of an earlier version of the paper and their valuable comments. We also thank the anonymous referees for their helpful comments.

References

1. Anderson, J.H., Kim, Y.-J.: Fast and Scalable Mutual Exclusion. In: Jayanti, P. (ed.) DISC 1999. LNCS, vol. 1693, pp. 180–195. Springer, Heidelberg (1999)
2. Anderson, J.H., Kim, Y.-J.: An improved lower bound for the time complexity of mutual exclusion. Dist. Comp. 15 (2002)
3. Anderson, J.H., Kim, Y.-J., Herman, T.: Shared-memory mutual exclusion: major research trends since 1986. Dist. Comp. 16 (2003)
4. Anderson, T.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. 1 (1990)
5. Aspnes, J.: Randomized protocols for asynchronous consensus. Dist. Comp. 16(2-3) (2003)
6. Attiya, H., Hendler, D., Woelfel, P.: Tight RMR lower bounds for mutual exclusion and other problems. In: 40th STOC (2008)
7. Bender, M.A., Gilbert, S.: Mutual exclusion with $o(\log^2 \log n)$ amortized work. In: 52nd FOCS (2011)
8. Danek, R., Golab, W.: Closing the complexity gap between mutual exclusion and fcfs mutual exclusion. In: 27th PODC (2008)
9. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Communications of the ACM 8 (1965)
10. Giakkoupis, G., Woelfel, P.: Tight rmr lower bounds for randomized mutual exclusion. In: 44th STOC (to appear, 2012)
11. Golab, W., Hadzilacos, V., Hendler, D., Woelfel, P.: Constant-rmr implementations of cas and other synchronization primitives using read and write operations. In: 26th PODC (2007)
12. Golab, W., Higham, L., Woelfel, P.: Linearizable implementations do not suffice for randomized distributed computation. In: 43rd STOC (2011)
13. Hendler, D., Woelfel, P.: Adaptive randomized mutual exclusion in sub-logarithmic expected time. In: 29th PODC (2010)
14. Hendler, D., Woelfel, P.: Randomized mutual exclusion with sub-logarithmic rmr-complexity. Dist. Comp. 24(1) (2011)
15. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12 (1990)
16. Jayanti, P.: Adaptive and efficient abortable mutual exclusion. In: 22nd PODC (2003)
17. Kim, Y.-J., Anderson, J.H.: A Time Complexity Bound for Adaptive Mutual Exclusion (Extended Abstract). In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, p. 1. Springer, Heidelberg (2001)
18. Kim, Y.-J., Anderson, J.H.: Adaptive mutual exclusion with local spinning. Dist. Comp. 19 (2007)
19. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comp. Syst. 9 (1991)
20. Pareek, A., Woelfel, P.: RMR-efficient randomized abortable mutual exclusion. arXiv:1208.1723 (2012)
21. Scott, M.: Non-blocking timeout in scalable queue-based spin locks. In: 21st PODC (2002)
22. Yang, J., Anderson, J.: A fast, scalable mutual exclusion algorithm. Dist. Comp. 9 (1995)

Abortable Reader-Writer Locks Are No More Complex Than Abortable Mutex Locks

Prasad Jayanti and Zhiyu Liu

Department of Computer Science, Dartmouth College,
Hanover, New Hampshire
prasad@cs.dartmouth.edu, Zhiyu.Liu.GR@dartmouth.edu

Abstract. When a process attempts to acquire a mutex lock, it may be forced to wait if another process currently holds the lock. In certain applications, such as real-time operating systems and databases, indefinite waiting can cause a process to miss an important deadline [19]. Hence, there has been research on designing *abortable* mutual exclusion locks, and fairly efficient algorithms of $O(\log n)$ RMR complexity have been discovered [11,14] (n denotes the number of processes for which the algorithm is designed).

The abort feature is just as important for a reader-writer lock as it is for a mutual exclusion lock, but to the best of our knowledge there are currently no abortable reader-writer locks that are starvation-free. We show the surprising result that any abortable, starvation-free mutual exclusion algorithm of RMR complexity $t(n)$ can be transformed into an abortable, starvation-free reader-writer exclusion algorithm of RMR complexity $O(t(n))$. Thus, we obtain the first abortable, starvation-free reader-writer exclusion algorithm of $O(\log n)$ RMR complexity. Our results apply to the Cache-Coherent (CC) model of multiprocessors.

Keywords: concurrent algorithm, synchronization, reader-writer exclusion, mutual exclusion, abortability, RMR complexity, shared memory algorithm.

1 Introduction

1.1 Reader-Writer Exclusion

Mutual Exclusion, where n asynchronous processes share a resource that can be accessed by only one process at a time, is a fundamental problem in distributed computing [7]. In the standard formulation of this problem, each process repeatedly cycles through four sections of code—the *Remainder*, *Try*, *Critical*, and *Exit* Sections. The process stays in the *Remainder Section* as long as it is not interested in the resource. When it becomes interested, it executes the *Try Section* to compete with other processes for the access to resource. The process then enters the *Critical Section* (CS), where it accesses the resource. Finally, to relinquish its right over the resource, the process executes the *Exit Section* and moves back to the Remainder Section. The *mutual exclusion problem* is to design the Try and Exit Sections so that at most one process is in the CS at any time. The Try and Exit Sections are normally thought of as the acquisition and the release of an *exclusive lock* to the resource.

Reader-Writer Exclusion is an important and natural generalization of mutual exclusion. This problem was first formulated and solved over forty years ago by Courtois, Heymans, and Parnas [6] and continues to receive much research attention [17,10,3,15,5,4]. Here the shared resource is a buffer and processes are divided into *readers* and *writers*. If a writer is in the CS, no other process may be in the CS at that time. However, since readers do not modify the buffer, the exclusion requirement is relaxed for readers: any number of readers are allowed to be in the CS simultaneously. A reader-writer exclusion algorithm takes advantage of this relaxation in the exclusion requirement, besides satisfying several other desirable properties, such as concurrent entering, first-in-first-enabled among the readers, first-come-first-served among the writers, and bounded-exit. These properties are defined later in Section 2.

When readers and writers compete for the CS, the algorithm has three natural choices: (i) readers have higher priority, (ii) writers have higher priority, or (iii) neither class has a higher priority and no reader or writer starves. In this paper we consider only the last (starvation-free) case.

1.2 Remote Memory Reference (RMR) Complexity

In an algorithm that runs on a multiprocessor, if a processor p accesses a shared variable that resides at p's local memory module, the access will be fast, but if p accesses a remote shared variable, the access can be extremely slow (because of the delay in gaining exclusive access to the interconnection bus and the high latency of the bus). Research in the last two decades has therefore been driven by the goal to minimize the number of *remote memory references (RMRs)* (see the survey [1]). (In *Distributed Shared Memory* (DSM) machines, a reference to a shared variable X is considered remote if X is at a memory module of a different processor; in *Cache Coherent* (CC) machines, a reference to X by a process p is considered remote if X is not in p's cache.) This goal implies that algorithms should be designed to achieve local spinning, i.e., processes do not make any remote references in busywait loops. The ideal goal is to design locking algorithms whose *RMR complexity*–the worst case number of remote memory references made by a process to enter and exit the CS once–is a constant, independent of the number of processes executing the algorithm. This goal was achieved for mutual exclusion over twenty years ago—Anderson's algorithm achieves constant RMR complexity for CC machines [2], and Mellor-Crummey and Scott's algorithm achieves constant RMR complexity for both CC and DSM machines [16].

In contrast, for the reader-writer problem, constant RMR complexity is achievable for CC machines [3,4], but is provably impossible for DSM machines: Danek and Hadzilacos' lower bound proof for 2-Session Group Mutual Exclusion implies that a sublinear RMR complexity algorithm satisfying concurrent entering is impossible for the reader-writer exclusion problem [10].

1.3 Abortability

In certain applications, such as real-time operating systems and databases, indefinite waiting can cause a process to miss an important deadline [19]. Therefore, there has been a lot of research on the design of exclusion algorithms that provide an extra feature—the *Abort Section*—that a busywaiting process in the Try Section can execute if it wishes to quit the protocol [19,18,11,14]. Since a process executes the Abort Section only when it cannot afford to wait any further, it is imperative that this section of code be *wait-free*, i.e., a process completes the Abort Section in a bounded number of its own steps, regardless of how its steps interleave with the steps of other processes.

For the mutual exclusion problem, efficient abortable algorithms are known: Jayanti's algorithm has $O(\log n)$ RMR complexity, where n is the number of processes [11]. Lee's algorithm has the same worst-case complexity, but achieves $O(1)$ complexity for the case where no process aborts [14]. For the reader-writer exclusion problem, Zheng designed an abortable algorithm of $O(n)$ RMR complexity, but this algorithm applies only for the reader-priority case [20]. To the best of our knowledge, there is no abortable algorithm for the starvation-free case, which is the focus of this paper.

1.4 The Main Result

We investigate the hardness of the abortable reader-writer exclusion problem relative to the abortable mutual exclusion problem. Let $me(n)$ and $rw(n)$ denote the worst case RMR complexities of the abortable, starvation-free, mutual exclusion problem and the abortable, starvation-free reader-writer exclusion problem, respectively. Since mutual exclusion is a special case of reader-writer exclusion where all processes act as writers, it follows that $me(n) \leq rw(n)$, i.e., $me(n) = O(rw(n))$. Is the converse true? To our surprise, we found the answer is yes. Specifically, we present a constant RMR complexity transformation that converts any abortable, starvation-free mutual exclusion algorithm into an abortable, starvation-free reader-writer exclusion algorithm. This establishes that $rw(n) \leq me(n) + O(1)$, and thus $rw(n) = O(me(n))$.

Our result has two significant implications:

- It establishes that $rw(n) = \Theta(me(n))$, i.e., abortable, starvation-free, reader-writer exclusion is exactly as hard as abortable, starvation-free, mutual exclusion.
- Our transformation, when applied to the $O(\log n)$ abortable mutual exclusion algorithm, gives rise to an abortable, starvation-free reader-writer exclusion algorithm of $O(\log n)$ RMR complexity. To the best of our knowledge, this is the first abortable, starvation-free reader-writer exclusion algorithm.

1.5 How the Transformation Is Structured

Our transformation is presented in two steps. First, in Section 3, we design an abortable reader-writer algorithm $\mathcal{A}$ that supports only a single writer. Then, in

Section 5, we show how to combine $\mathcal{A}$ with an abortable mutual exclusion lock $\mathcal{M}$ to obtain an abortable reader-writer algorithm that supports an arbitrary number of writers (and readers). The design of $\mathcal{A}$ in the first step constitutes the intellectual contribution of this paper. (The second step uses the simple idea that multiple writers compete for the lock $\mathcal{M}$ and the successful one proceeds to execute the single-writer algorithm $\mathcal{A}$.)

2 The Abortable Reader-Writer Exclusion Problem

In this section we provide a clear statement of the abortable reader-writer exclusion problem.

Each process has five sections of code—Remainder, Try, Critical, Exit, and Abort Sections. A process executes its code in phases. In each phase, the process does one of two things: (1) starts in the Remainder Section; then executes the Try Section, the Critical Section (CS), and the Exit Section (in that order); and then goes back to the Remainder Section, or (2) starts in the Remainder Section; then executes the Try Section, possibly partially; then executes the Abort Section; and then goes back to the Remainder Section.

The Try Section is divided into a *doorway*, followed by a *waiting room* [13]. It is required that the doorway be wait-free, i.e., each process completes the doorway in a bounded number of its steps, regardless of how its steps interleave with the steps of other processes.

We say a reader r in the Try Section is *enabled* if r will enter the CS in a bounded number of its own steps, regardless of how its steps interleave with the steps of other processes.

The *reader-writer exclusion problem* is to design the Try, Exit, and Abort Sections of code for each process so that the following properties hold in all runs:

– (P1) Reader-Writer Exclusion: If a writer is in the CS, then no other process is in the CS at that time.
– (P2) Bounded Abort: Each process completes the Abort Section in a bounded number of its steps, regardless of how its steps interleave with the steps of other processes.
– (P3) Concurrent Entering: Since readers don't conflict with each other, it is desired that they do not obstruct each other from entering the CS. More specifically, if all writers are in the Remainder Section and will remain there, then every reader in the Try Section enters the CS in a bounded number of its own steps [9,12].
– (P4) FIFE among readers: Unlike writers that may only access the CS one at a time, any number of readers can cohabit the CS. Consequently, if a reader r completes the doorway before another reader r' enters the doorway, there is no reason to delay the entry of r' into the CS for the sake of r. We use the *First-In-First-Enabled* (FIFE) property, first defined by Fischer et al for the k-exclusion problem [8], to define fairness among readers, as follows:

If a reader r in the Try Section completes the doorway before another reader r' enters the doorway, and r' subsequently enters the CS, then one of the following three conditions holds: (i) r enters the CS before r' enters the CS, or (ii) r begins executing the Abort Section before r' enters the CS, or (iii) r is enabled to enter the CS when r' enters the CS.

– (P5) Starvation-Freedom: When readers and writers compete for the CS, the algorithm has three natural choices: (i) give higher priority to readers, (ii) give higher priority to writers, or (iii) treat both classes of processes fairly. In this paper we consider only the third case and require the *starvation-freedom* property: if a process in the Try Section does not abort, it will eventually enter the CS, under the assumption that no process stops taking steps in the Try, Exit, or Abort Sections and no process stays in the CS forever.
– (P6) Bounded Exit: Every process completes the Exit Section in a bounded number of its own steps.
– (P7) FCFS among writers: We use the *First-Come-First-Served (FCFS)* property, first defined by Lamport for the mutual exclusion problem [13], to define fairness among writers, as follows: If a writer w completes the doorway before a writer w' enters the doorway and w does not abort, then w' does not enter the CS before w.

3 Single-Writer Multi-reader Algorithm

Figure 1 shows our abortable single-writer multi-reader algorithm. The subroutine R-Abort is what readers execute when they abort from Line 2, while W-Abort-L7 and W-Abort-L11 are what the writer executes when it aborts from Line 7 and Line 11 respectively. W-Abort-L7 is empty, meaning the writer can quit immediately if it aborts from Line 7. It is worth noting that this algorithm works for any number of readers.

This algorithm employs four shared variables that support read/write and fetch&add (F&A)[1] operations. To fully understand the algorithm, we first need to know the purposes of these shared variables.

– G: Intuitively, G is an entrance that has two "gates", Gate 0 and Gate 1, through which readers enter the CS. When the writer is requesting to enter the CS or occupying the CS, the algorithm has either $G = 0$ or $G = 1$, indicating either only Gate 0 or only Gate 1 is open. This "gates" idea was proposed by Bhatt and Jayanti [3]. Here, in order to support aborting, we make the following modification: if the writer is not requesting to or occupying the CS, the algorithm has $G \geq 2$, indicating both gates are open.
– X: X has two components. Let us call them $X.1$ and $X.2$ from left to right. $X.1$ is a single bit read by readers to figure out which gate they should pass. $X.2$ is a counter for the number of readers that are requesting to enter or

[1] People usually assume that fetch&add returns the previous value of the variable. But for convenience, we assume in this paper that it returns the new value: we assume the operation $F\&A(X, a)$ returns $(x + a)$, not x, where x is the previous value of X.

```
G: integer
Flag: single bit
X: pair of integers, where the first component only consists of one bit
Y: integer
Initially, $X = [0, 0]$, $Y = 0$, $Flag = 0$, $G = 2$

        Reader
1.      $[g, -] \leftarrow F\&A(X, [0, 1])$
2.      wait till $G \geq 2 \vee G = g$
3.      CS
        do R-Exit

        R-Exit
4.      $[g', -] \leftarrow F\&A(X, [0, -1])$
        if $g' \neq g$
5.          $a \leftarrow F\&A(Y, 1)$
            if $(a = 0)$
6.              $F\&A(Flag, 1)$

        R-Abort
        do R-Exit

        Writer
7.      wait till $Flag = 0$
8.      $F\&A(G, -2)$
9.      $[s, r] \leftarrow F\&A(X, [1, 0])$
10.     $c \leftarrow F\&A(Y, -r)$
        if $c \neq 0$
11.         wait till $Flag = 1$
12.     CS
        do R-Exit

        W-Exit
13.     $G \leftarrow s + 2$
        if $(c \neq 0)$
14.         $F\&A(Flag, 1)$

        W-Abort-L7
        do nothing

        W-Abort-L11
        do W-Exit
```

Fig. 1. Abortable Single-Writer Multi-Reader Algorithm

currently in the CS[2]: When a reader enters the Try Section, it first increments $X.2$ by 1 at Line 1. When a reader exits, it first decrements $X.2$ by 1 at Line 4. Therefore, $X.2$ is equal to the number of readers that are requesting to enter or currently in the CS. When the writer wants to access the CS, it reads $X.2$ (i.e., increments $X.2$ by 0) at Line 9 to know the number of readers that are in the CS or about to enter the CS. The writer then waits at Line 11 until all such readers have exited the CS.

– Y: Y is what the writer uses to communicate with readers about how many readers it needs to wait for. Intuitively, $-Y$ is the number of readers the writer is waiting for when the writer is at Line 11. As we mentioned above, when the writer executes Line 9, it gets from $X.2$ the number of readers that are in the CS or about to enter the CS—the number of readers the writer has to wait for. Then the writer writes this information into Y by subtracting that number from Y at Line 10. We will explain later that only the readers

[2] Our algorithm needs to do $F\&A$ on both components simultaneously. It is easy to implement this operation in $O(1)$ RMR. For example, to implement $F\&A(X, [a, b])$, we can simply do $F\&A(X, a * 2^k + b)$, where k is the length of $X.2$. $X.1$ is a single bit in our algorithm. Hence, if $X.1 = 1$ and we increment it by 1, it will become 0. This is also easy to implement. For example, we can set $X.1$ to be the rightmost bit of X, or we can simply return the mod 2 value of $X.1$ when we read it.

that the writer needs to wait for will find $g' \neq g$ at Line 4 and therefore go to Line 5 to increment Y by 1 each time. Thus, $-Y$ will always be equal to the number of readers the writer is still waiting for. When the last of such readers executes Line 5, it will find $a = 0$, i.e., $Y = 0$, indicating all the readers the writer waits for have exited. Then this reader will wake up the writer by executing Line 6.

– *Flag*: *Flag* is a single bit indicating what the writer should do. As we mentioned before, when the writer is waiting for some readers to leave the CS at Line 11, the last of such readers will go to Line 6 to increment *Flag*. Thus, when the writer finds $Flag = 1$, it knows that it can now enter the CS. When the writer exits, it flips *Flag* back to 0 by executing Line 14. On the other hand, if the writer does not want to wait any longer at Line 11, it can abort and then execute Line 14 to increment *Flag* to 1. When the writer enters the Try Section again, some slow readers may not have exited yet. In this case, the writer will find $Flag = 1$ at Line 7 because of the previous increment at Line 14, so the writer needs to wait at Line 7 until all such readers have exited. When the last of such readers finally leaves, it increments *Flag* at Line 6 so as to set *Flag* back to 0. Thus, the writer knows that it can now move on to request its access to the CS.

With the above description of the shared variables, we can now explain the details of the algorithm. We begin by describing how readers get permission to enter the CS. Let us start with the initial configuration. Suppose a reader first enters the Try Section. It first executes Line 1, reading the first component of X by incrementing $X.1$ by 0. Since $X.1 = 0$, the reader needs to go through Gate 0. Initially $G = 2$, which indicates both gates are open. Therefore, when the reader executes Line 2, it will find $G\,2$. Hence, the reader is enabled to enter the CS.

If it is the writer that first enters the Try Section, it will find $Flag = 0$ at Line 7, go to Line 8, and change G from 2 to 0. If a reader now enters the Try Section and executes Line 1 and Line 2, it will find $g = X.1 = G = 0$. That is, the reader needs to pass through Gate 0 and Gate 0 is open now. Therefore, the reader can also enter the CS in this case.

If the writer executes Line 9 before a reader executes Line 1, the reader needs to pass through Gate 1, since the operation of Line 9 flips $X.1$ to 1. Then, when the reader goes to Line 2, it will find $G = 0$ and $g = 1$. Therefore, the reader has to wait at Line 2. Hence, Line 9 becomes a critical point for synchronization: if a reader executes Line 1 before the writer executes Line 9, it is *enabled* to enter the CS; otherwise, the reader has to wait and let the writer enter the CS first.

Now suppose a reader executed Line 1 after the writer executed Line 9 and the reader is waiting at Line 2. When the writer leaves the CS or aborts from the Try Section (i.e., from Line 11), it executes Line 13, setting $G = X.1 + 2 = 1 + 2 = 3$ (since the writer stored $X.1$'s value in s). This implies that both gates are open and the reader is enabled now.

Next time the writer enters the Try Section, it will change G from 3 to 1 at Line 8 and hence $X.1 = G = 1$. Therefore, Line 9 is still a critical point for synchronization: if a reader executes Line 1 before the writer executes Line 9, it will find $g = G = 1$ and hence is enabled; otherwise, the reader will find $g = 0$ and $G = 1$, and it will wait at Line 2 until the writer executes Line 13. After the writer executes Line 13, $G = 2$ and $X.1 = 0$. Then G and $X.1$ are not set back to their initial value, and the scenario is now the same as that in the initial configuration.

There is one risk for readers: when the writer is in the Remainder Section, a reader R may execute Line 1 and fall asleep before executing Line 2. The writer now enters the Try Section and aborts, making $G \geq 2$ but $G \neq R.g + 2$. Then the writer enters the Try Section again, executing Line 8 to make $G < 2$ and $G \neq R.g$. If R now wakes up, it will be blocked from entering the CS, which we do not expect to happen. However, this situation cannot happen: the second time the writer enters the Try Section, it will find $Flag = 1$ at Line 7 and hence it cannot execute Line 8 to cause that risk. We will explain the reason later.

When a reader aborts from Line 2, it simply does its Exit Section. If the writer is still in the CS, or is in the Remainder Section after executing the Exit Section, the aborting reader will find $X.1$ unchanged at Line 4 since it executed Line 1. Thus, the reader has $g = g'$ and then leaves without executing Line 5, just like it has not entered the Try Section. On the other hand, if the writer has exited, entered the Try Section again, and changed $X.1$ at Line 9, then the aborting reader will find $g \neq g'$ after executing Line 4. Hence, the reader just executes Line 5 and probably Line 6, just as a non-aborting reader exits from the CS. This is what we want the reader to do, since this is a situation where the writer thinks the reader is enabled and the writer waits until the reader leaves.

Let us now explain how the writer gets access to the CS. Initially, $Flag = 0$. Therefore, the writer will find $Flag = 0$ the first time it enters the Try section. Then, the writer can execute Line 8 and Line 9, claiming its request to access the CS. As we mentioned earlier, any readers that enter the Try Section after the writer makes its request (at Line 9) will be blocked from getting into the CS. On the other hand, that enter the Try Section before the writer makes its request are already enabled, and the writer has to wait until all these readers exit. At Line 9, the writer simultaneously gets from $X.2$ the number of the readers it needs to wait for. Then, at Line 10, the writer subtracts from Y the number of such readers, r. Note that all the readers that entered the Try Section before the writer executed Line 9 have $g \neq X.1$ now because of the change on $X.1$ at Line 9. Therefore, if these readers exit or abort (these two subroutines are actually the same), they will find $g' = X.1 \neq g$ at Line 4 and hence execute Line 5 to increment Y. On the other hand, if any blocked reader waiting at Line 2 aborts (or finally enters and then leaves the CS after the writer has left), it will find $g' = g$ because it executed Line 1 after the writer made the change to Y at Line 9. Therefore, this reader will leave without touching Y. That is, only the readers that the writer needs to wait for can find $g' \neq g$ at Line 4 and increment Y at Line 5.

Hence, when the writer gets the value c at Line 10 after subtracting r from Y, it knows that there are exactly $-c$ readers it has to wait for. If $c \neq 0$, the writer will wait until the last of such readers increments Y to 0 at Line 5 and sets $Flag$ to 1 at Line 6. When the writer exits, it sets the single bit $Flag$ back to 0 by incrementing $Flag$ by 1 at Line 14, so that it will find $Flag = 0$ to pass Line 7 next time it enters the Try Section. On the other hand, if the writer doesn't want to wait any longer and decides to abort from Line 11, it will also increment $Flag$ by 1. Thus, if the last of enabled readers has not exited or aborted yet to execute Line 6, we have $Flag = 1$. Therefore, when the writer enters the Try Section again, it has to wait at Line 7 until $Flag = 0$, implying the last reader has finally executed Line 6 to flip $Flag$ from 1 to 0.

If the writer finds $c = 0$ after executing Line 10, it knows that all enabled readers have executed Line 5 before it executes Line 10. That is, all the reader it needs to wait for have exited the CS. Hence, the writer can enter the CS immediately. An important observation is that, if a reader executed Line 5 before the writer executes Line 10 to decrement Y, it incremented Y to a positive value and hence it would find $a > 0$ at Line 5. Therefore, in the case where the writer has $c = 0$, i.e., all enabled readers have exited before the writer requests to access the CS, no reader can get into Line 6 to change $Flag$. Hence, the writer can just leave $Flag = 0$ unchanged by not executing Line 14 when it has $c = 0$. Thus, it will find $Flag = 0$ at Line 7 next time it enters the Try Section.

Now consider that the writer aborted last time and comes into the Try Section again and finds $Flag = 1$ at Line 7. As we discussed earlier, this implies that some readers the writer needed to wait for last time have not exited yet. Thus, the writer has to wait at Line 7 until all such readers have left. If the writer wants to abort from Line 7, it does nothing and simply leaves. But next time it enters the Try Section, it still needs to wait there until $Flag = 0$. It is easy to figure out that, if any reader enters and then leaves during the period when the writer is in the Remainder Section or waits at Line 7, it will find $g = g'$ at Line 4. Hence, such readers cannot change Y or $Flag$. Therefore, the writer does not need to worry about the risk that some reader updates Y or $Flag$ improperly. After the writer passes Line 7, it can then request the CS by executing the Try Section, just as it did for the first time.

4 Correctness of Single-Writer Multi-reader Algorithm

Our proof is invariant-based. In Figure 2, we present the invariant $\mathcal{I}$ that our single-writer multi-reader algorithm satisfies. We prove $\mathcal{I}$ always holds by showing that $\mathcal{I}$ holds in the initial configuration, and that, if $\mathcal{I}$ holds in any reachable configuration $\mathcal{C}$, it still holds after any process takes a step. Due to space constraints, we refer interested readers to the full version of the paper for the proof of the invariant.

It is obvious that this single-writer multi-reader algorithm only employs $O(1)$ shared objects and satisfies Bounded Abort and Bounded Exit properties. In the following, we prove that this algorithm satisfies properties P1 and P3–P5, and that it achieves $O(1)$ RMR complexity in CC models.

- Definitions:
  1. R and W denote a reader and the writer, respectively.
  2. $\mathcal{I}_i$ is a collection of predicates that are true when the writer's program counter is at Line i.
  3. $PC_R = i$ states that reader R's program counter is at Line i.
  4. $X.1$ denotes the first field of X. Similar definitions for $X.2$, $Y.1$, etc.
- $\mathcal{I}_G$ (global invariant):
  1. $\forall R, PC_R \in \{2,3,4,5,6\} \Longrightarrow (R.g = 1 \vee R.g = 0)$
  2. $X.2 = |\{R|\ PC_R \in \{2,3,4\}\}|$

- $\mathcal{I}_7$:
  1. $G = X.1 + 2$
  2. $Y = -|\{R|\ PC_R \in \{2,3,4,5\} \wedge R.g \neq X.1\}|$
  3. $\forall R, PC_R = 5 \Longrightarrow R.g \neq X.1$
  4. $Y \neq 0 \Longrightarrow (Flag = 1 \wedge |\{R|PC_R = 6\}| = 0)$
  5. $(Flag = 1 \wedge Y = 0) \Longrightarrow |\{R|PC_R = 6\}| = 1$
  6. $Flag = 0 \Longrightarrow (Y = 0 \wedge |\{R|PC_R = 6\}| = 0)$
- $\mathcal{I}_8$:
  1. $G = X.1 + 2$
  2. $Y = 0$
  3. $|\{R|\ PC_R \in \{2,3,4\} \wedge R.g \neq X.1\}| = 0$
  4. $|\{R|\ PC_R \in \{5,6\}\}| = 0$
  5. $Flag = 0$
- $\mathcal{I}_9$:
  1. $G = X.1$
  2. $Y = 0$
  3. $|\{R|\ PC_R \in \{2,3,4\} \wedge R.g \neq X.1\}| = 0$
  4. $|\{R|\ PC_R \in \{5,6\}\}| = 0$
  5. $Flag = 0$
- $\mathcal{I}_{10}$:
  1. $G = 1 - X.1$
  2. $Y = W.r - |\{R|\ PC_R \in \{2,3,4,5\} \wedge R.g \neq X.1\}| \geq 0$
  3. $\forall R, PC_R = 3 \Longrightarrow R.g \neq X.1$
  4. $\forall R, PC_R = 5 \Longrightarrow R.g \neq X.1$
  5. $|\{R|\ PC_R \in 6\}| = 0$
  6. $Flag = 0$
  7. $W.s = X.1$
- $\mathcal{I}_{11}$:
  1. $G = 1 - X.1$
  2. $Y = -|\{R|\ PC_R \in \{2,3,4,5\} \wedge R.g \neq X.1\}| \leq 0$
  3. $\forall R, PC_R = 3 \Longrightarrow R.g \neq X.1$
  4. $\forall R, PC_R = 5 \Longrightarrow R.g \neq X.1$
  5. $Y \neq 0 \Longrightarrow Flag = 0 \wedge |\{R|PC_R = 6\}| = 0$
  6. $(Flag = 0 \wedge Y = 0) \Longrightarrow |\{R|PC_R = 6\}| = 1$
  7. $Flag = 1 \Longrightarrow (Y = 0 \wedge |\{R|PC_R = 6\}| = 0)$
  8. $W.s = X.1$
  9. $W.c \neq 0$
- $\mathcal{I}_{12}$:
  1. $G = 1 - X.1$
  2. $Y = |\{R|\ PC_R \in \{2,3,4,5\} \wedge R.g \neq X.1\}| = 0$
  3. $|\{R|PC_R \in \{3,5,6\}\}| = 0$
  4. $Flag = 0 \Longleftrightarrow W.c = 0$
  5. $W.s = X.1$
- $\mathcal{I}_{13}$:
  1. $G = 1 - X.1$
  2. $Y = -|\{R|\ PC_R \in \{2,3,4,5\} \wedge R.g \neq X.1\}| \leq 0$
  3. $\forall R, PC_R = 3 \Longrightarrow R.g \neq X.1$
  4. $\forall R, PC_R = 5 \Longrightarrow R.g \neq X.1$
  5. $W.c = 0 \Longrightarrow Y = 0$
  6. $W.c = 0 \Longrightarrow |\{R|\ PC_R = 6\}| = 0$
  7. $W.c = 0 \Longrightarrow Flag = 0$
  8. $Y \neq 0 \Longrightarrow (Flag = 0 \wedge |\{R|PC_R = 6\}| = 0)$
  9. $(W.c \neq 0 \wedge Flag = 0 \wedge Y = 0) \Longrightarrow |\{R|PC_R = 6\}| = 1$
  10. $(W.c \neq 0 \wedge Flag = 1) \Longrightarrow (Y = 0 \wedge |\{R|PC_R = 6\}| = 0)$
  11. $W.s = X.1$
- $\mathcal{I}_{14}$:
  1. $G = X.1 + 2$
  2. $Y = -|\{R|\ PC_R \in \{2,3,4,5\} \wedge R.g \neq X.1\}| \leq 0$
  3. $\forall R, PC_R = 5 \Longrightarrow R.g \neq X.1$
  4. $Y \neq 0 \Longrightarrow (Flag = 0 \wedge |\{R|PC_R = 6\}| = 0)$
  5. $(Flag = 0 \wedge Y = 0) \Longrightarrow |\{R|PC_R = 6\}| = 1$
  6. $Flag = 1 \Longrightarrow (Y = 0 \wedge |\{R|PC_R = 6\}| = 0)$

Fig. 2. Invariant $\mathcal{I}$ of the Abortable Single-Writer Multi-Reader Algorithm

Lemma 1. *This algorithm satisfies reader-writer exclusion.*

Proof of Lemma 1: $\mathcal{I}_{12}$ states that when the writer is in the CS (i.e., at Line 12), no reader is in the CS (i.e., at Line 3). Hence the proof. □

Lemma 2. *This algorithm satisfies starvation freedom.*

The proof of this lemma is based on the following claim.

Claim. If a reader R is in the Try Section and has $G \geq 2 \vee G = R.g$ at some time t, then R is enabled after t, and $G \geq 2 \vee G = R.g$ holds until R has exited.

Proof of Claim 4: Suppose we have $G \geq 2 \vee G = R.g$ at time t, for a reader R with $PC_R = 2$. If G remains unchanged, R will be able to get into the CS after executing Line 2. Note that G's value can only be changed at Line 8 and Line 13. According to $\mathcal{I}_{13}$, when W is about to execute Line 13, $W.s = X.1 \in \{0, 1\}$. Hence, $G \geq 2$ after Line 13 is executed. Thus, R can still get into the CS. According to $\mathcal{I}_9$, we have $|\{R|\ PC_R = 2 \wedge R.g \neq X.1\}| = 0$ and $G = X.1$ after W executes Line 8. Therefore, we know the reader R must have $R.g = X.1 = G$. Thus, R still satisfies $G = R.g$ and hence can still get into the CS. Therefore, R is enabled after t. Moreover, $\mathcal{I}_9$ also states that $|\{R|\ PC_R \in \{3, 4\} \wedge R.g \neq X.1\}| = 0$ and $|\{R|\ PC_R \in \{5, 6\}\}| = 0$. The former predicate implies that, if R is at Lines 3–4, $R.g = X.1 = G$ after W executes Line 8. The latter one implies that W cannot execute Line 8 when R is at Lines 5–6. Therefore, $G \geq 2 \vee G = R.g$ always holds until R has exited. Hence the claim. □

Proof of Lemma 2: First we prove the writer W will not starve. Since W can wait at Line 7 and Line 11, we need to prove W will not wait forever at either of these two lines. We will analyze all possibilities below.

1.1) Suppose W waits at Line 7 and $Flag = 0$. Then according to $\mathcal{I}_7$, $|\{R|\ PC_R = 6\}| = 0$ as long as W is at Line 7. Since the only way for readers to change $Flag$ is to execute Line 6, we know $Flag$ will remain 0. Hence, W can pass Line 7 after W takes a step at any time. **1.2)** Suppose W waits at Line 7 and $Flag = 1$. **1.2.a)** If $Y = 0$, then, by $\mathcal{I}_7$, we have $|\{R|\ PC_R = 6\}| = 1$. Therefore, after the only reader at Line 6 takes a step, we have $Flag = 0$ and hence W can pass Line 7 by taking a step. **1.2.b)** If $Y \neq 0$, then, by $\mathcal{I}_7$, we have $Y = -|\{R|\ PC_R \in \{2, 3, 4, 5\} \wedge R.g \neq X.1\}|$, $|\{R|\ PC_R = 6\}| = 0$, and $G = X.1 + 2 \geq 2$. Since $X.1$ remains unchanged as long as W waits at Line 7, any reader $R \in \{R|\ PC_R \in \{2, 3, 4, 5\} \wedge R.g \neq X.1\}$ will finally go to Line 5 and increment Y. $\mathcal{I}_7$ also states $\forall R, PC_R = 5 \Longrightarrow R.g \neq X.1$. This implies that all readers R' with $R'.g = X.1$ must be at Line 2–4. Hence, these readers will find $R'.g = R'.g'$ after executing Line 4 and cannot get into Line 5 to change Y. On the other hand, any new reader R' will get $R'g = X.1$ after executing Line 1. Therefore, we can conclude that all the readers that can affect Y must be those that are already at Line 2–5 with $R.g \neq X.1$ until W waits at Line 7. Since $Y = -|\{R|\ PC_R \in \{2, 3, 4, 5\} \wedge R.g \neq X.1\}|$, Y will become 0 after all these readers execute Line 5. Now, we get back to case **1.2.a)** and know that W can pass Line 7.

2) Suppose W waits at Line 11. By similar arguments, we can prove the following facts based on the invariant $\mathcal{I}$.

- **2.1)** Once $Flag$ becomes 1, it will remain 1 until W goes through Line 11. Thus, W will not wait at Line 11 forever.
- **2.1.a)** If $Flag = 0$ and $Y = 0$, then there is only one reader at Line 6 and $Flag$ will become 1 after that reader takes a step.

– **2.1.b)** If $Flag = 0$ and $Y \neq 0$, then after all the current readers R's with $R.g \neq X.1$ finally execute Line 5, we have $Y = 0$ and know that $Flag$ will become 1.

According to these facts, we can conclude that W will not wait at Line 11 forever. Hence, W won't starve.

Since W won't starve, W can finally execute Line 13 and make $G \geq 2$. Thus, any reader R waiting at Line 2 will then find $G \geq 2 \vee G = R.g$. By Claim 4, R is now enabled and hence won't starve.

Lemma 3. *This algorithm satisfies FIFE among readers.*

Proof of Lemma 3: The doorway for the readers is line 1. Assume this algorithm does not satisfy FIFE. That is, there exists a scenario where a reader R executes Line 1 before another reader R' does, and R' is enabled earlier than R. Since R comes into the Try Section before R', at the moment R' becomes enabled, R and R' must be both in the Try Section, i.e., $PC_R = PC'_R = 2$.

If $R.g = R'.g$, then at the moment R' is enabled, we know $R'.g = R.g = G$. By Claim 4, this implies that R is also enabled at the same time, contradicting the assumption that R' is enabled earlier than R.

Suppose $R.g \neq R'.g$. Since R and R' get the values of $R.g$ and $R'g$ respectively from $X.1$ at Line 1, and $X.1$ can only be changed at Line 9, we can conclude that the writer W executes line 9 to flip $X.1$ at a time t between the time R executes line 1 and the time R' executes line 1. Since $X.1$ is a two-valued variable and $G = X.1$ holds when W is about to execute Line 9, this implies that $G = R.g$ holds either before t or after t. By Claim 4, R is enabled once $G \geq 2 \vee G = R.g$. Therefore, R is already enabled even before R' enters the Try Section, a contradiction.

Hence, this algorithm satisfies FIFE. □

Lemma 4. *This algorithm satisfies concurrent entering.*

Proof of Lemma 4: If the writer W is in the Remainder Section, i.e., $PC_W = 7$, then, by $\mathcal{I}_7$, $G = X.1 + 2 \geq 2$. Therefore, for all readers R at line 2, $G \geq 2 \vee G = R.g$. By Claim 4, This implies that all readers in the Try Section are enabled. Therefore, if W stays in the Remainder Section, every reader can enter the CS in a bounded number of its own steps. Hence, this algorithm satisfies concurrent entering. □

Lemma 5. *This algorithm has $O(1)$ RMR complexity in the CC model.*

Proof of Lemma 5: As we argued in the proof of Lemma 2, if W spins at Line 7 or Line 11, then $Flag$ can be changed at most once by readers, and after that change $Flag$ will remain a value such that W can pass Line 7 or Line 11. Combining this and the fact that all other lines for the writer are executed only once in a write attempt, we can conclude that the writer has $O(1)$ RMR complexity in the CC model.

Suppose a reader R spins at line 2. Since the writer cannot starve, it will execute Line 13 to make $G \geq 2$. As Claim 4 shows, R is now enabled and can get into the CS by taking one step. Moreover, G can only be updated by the writer. Therefore, it only takes $O(1)$ RMRs for R to spin at Line 2. Combining this and the fact that all other lines for readers are executed only once in a read attempt, we can conclude that the readers have $O(1)$ RMR complexity.

Hence, this algorithm has $O(1)$ RMR complexity in the CC model. □

Theorem 1. (Abortable Single-Writer Multi-Reader Starvation-Free Algorithm) *The algorithm in Figure 1 satisfies properties P1–P6. The RMR complexity of this algorithm in CC model is $O(1)$. This algorithm employs $O(1)$ number of shared variables that support read, write, and fetch&add operations.*

5 Transformation from Single-Writer Algorithm to Multi-Writer Algorithm

We convert our single-writer algorithm into a multi-writer algorithm using a simple transformation proposed by Bhatt and Jayanti [3](see Figure 3). Here readers simply execute the Reader procedure of the underlying abortable single-writer algorithm. Writers, on the other hand, first obtain an abortable mutex lock M and only then execute the Writer procedure of the underlying abortable single-writer algorithm. When a writer exits or aborts from the underlying Writer procedure, it releases the lock immediately after the last step of the Exit Section or Abort Section, respectively.

```
            M: abortable Mutex Lock

Reader                          Writer
SW-Reader()                     acquire(M)
                                SW-Writer()
                                release(M)
```

Fig. 3. Transforming a single-writer multi-reader algorithm *SW* to a multi-writer multi-reader algorithm

If M is an abortable mutex lock satisfying starvation-freedom, like the one in [11], we claim that the transformation in Figure 3 gives an abortable multi-writer multi-reader algorithm that satisfies P1–P6. More generally, given any abortable single-writer multi-writer algorithm, we can use this transformation to construct an abortable multi-writer multi-reader algorithm. This is trivially true because of two facts: (1) since the lock M ensures that only one writer accesses the underlying writer procedure at any time, the underlying single-writer multi-reader algorithm works as if it is in a single-writer multi-reader system; and (2) when a writer leaves the underlying writer procedure, it always releases the lock M so that the next writer can acquire M and then enter the underlying writer procedure, and hence no writer can starve if it doesn't abort. A formal proof can be found in [3].

What's more, if the lock M satisfies FCFS property, then it is obvious that the abortable multi-writer multi-reader algorithm satisfies FCFS among writers. Since our single-writer multi-reader algorithm only has $O(1)$ RMR complexity and employs $O(1)$ shared space, the RMR and shared space complexities of the multi-writer multi-reader algorithm depend on those of the abortable mutex lock M.

Theorem 2. (Abortable Multi-Writer Multi-Reader Starvation-Free Algorithm) *The algorithm in Figure 3 satisfies properties P1–P6. The RMR and shared space complexities of this algorithm are $O(r(M))$ and $O(s(M))$, where $r(M)$ and $s(M)$ are the RMR and shared space complexities of the abortable starvation-free mutex lock M used in this algorithm, respectively. If the M lock satisfies FCFS, then this algorithm satisfies FCFS among writers (property P7).*

Therefore, if the lock M used in Figure 3 is one of the $O(\log n)$-RMR locks in [11] and [14], we construct an abortable, starvation-free reader-writer lock of $O(\log n)$ RMR complexity. To our knowledge, this is the first abortable, starvation-free reader-writer lock.

References

1. Anderson, J.H., Kim, Y.-J., Herman, T.: Shared-memory mutual exclusion: major research trends since 1986. Distrib. Comput. 16(2-3), 75–110 (2003)
2. Anderson, T.E.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. 1(1), 6–16 (1990)
3. Bhatt, V., Jayanti, P.: Constant rmr solutions to reader writer synchronization. In: PODC 2010: Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, pp. 468–477. ACM, New York (2010)
4. Bhatt, V., Jayanti, P.: Specification and constant rmr algorithm for phase-fair reader-writer lock. In: ICDCN 2011: Proceedings of the 12th International Conference on Distributed Computing and Networking, pp. 119–130 (2011)
5. Brandenburg, B.B., Anderson, J.H.: Reader-writer synchronization for shared-memory multiprocessor real-time systems. In: ECRTS 2009: Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems, pp. 184–193. IEEE Computer Society, Washington, DC (2009)
6. Courtois, P.J., Heymans, F., Parnas, D.L.: Concurrent control with "readers" and "writers". Commun. ACM 14(10), 667–668 (1971)
7. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM 8(9), 569 (1965)
8. Fischer, M.J., Lynch, N.A., Burns, J.E., Borodin, A.: Resource allocation with immunity to limited process failure. In: SFCS 1979: Proceedings of the 20th Annual Symposium on Foundations of Computer Science, pp. 234–254. IEEE Computer Society, Washington, DC (1979)
9. Hadzilacos, V.: A note on group mutual exclusion. In: PODC 2001: Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, pp. 100–106. ACM, New York (2001)

10. Danek, R., Hadzilacos, V.: Local-Spin Group Mutual Exclusion Algorithms. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 71–85. Springer, Heidelberg (2004)
11. Jayanti, P.: Adaptive and efficient abortable mutual exclusion. In: PODC 2003: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, pp. 295–304. ACM, New York (2003)
12. Joung, Y.-J.: Asynchronous group mutual exclusion (extended abstract). In: PODC 1998: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, pp. 51–60. ACM, New York (1998)
13. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM 17(8), 453–455 (1974)
14. Lee, H.: Fast Local-Spin Abortable Mutual Exclusion with Bounded Space. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 364–379. Springer, Heidelberg (2010)
15. Lev, Y., Luchangco, V., Olszewski, M.: Scalable reader-writer locks. In: SPAA 2009: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures, pp. 101–110. ACM, New York (2009)
16. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. 9(1), 21–65 (1991)
17. Mellor-Crummey, J.M., Scott, M.L.: Scalable reader-writer synchronization for shared-memory multiprocessors. In: PPOPP 1991: Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 106–113. ACM, New York (1991)
18. Scott, M.L.: Non-blocking timeout in scalable queue-based spin locks. In: Proceedings of the 21st Annual Symposium on Principles of Distributed Computing, pp. 31–40 (2002)
19. Scott, M.L., Scherer III, W.N.: Scalable queue-based spin locks with timeout. In: Proceedings of the 8th Symposium on Principles and Practice of Parallel Programming, pp. 44–52 (2001)
20. Zheng, N.: Constant-rmr abortable reader-priority reader-writer algorithm. Technical Report TR2011-685, Dartmouth College, Computer Science, Hanover, NH (June 2011)

Pessimistic Software Lock-Elision

Yehuda Afek[1], Alexander Matveev[1], and Nir Shavit[2]

[1] Tel-Aviv University
{afek,matveeva}@post.tau.ac.il
[2] MIT and Tel-Aviv University
shanir@csail.mit.edu

Abstract. Read-write locks are one of the most prevalent lock forms in concurrent applications because they allow read accesses to locked code to proceed in parallel. However, they do not offer any parallelism between reads and writes.

This paper introduces *pessimistic lock-elision* (PLE), a new approach for non-speculatively replacing read-write locks with pessimistic (i.e. non-aborting) software transactional code that allows read-write concurrency even for contended code and even if the code includes system calls. On systems with hardware transactional support, PLE will allow failed transactions, or ones that contain system calls, to preserve read-write concurrency.

Our PLE algorithm is based on a novel encounter-order design of a fully pessimistic STM system that in a variety of benchmarks spanning from counters to trees, even when up to 40% of calls are mutating the locked structure, provides up to 5 times the performance of a state-of-the-art read-write lock.

Keywords: Multicore, Software Transactional memory, Locks, Lock-elision, Wait-free.

1 Introduction

Many modern applications make extensive use of read-write locks, locks that separate read-only calls from ones that can write. Read-write locks allow read-only calls, prevalent in many applications, to proceed in parallel with one another. However, read-write locks do not offer any parallelism between reads and writes.

In a ground breaking paper, Rajwar and Goodman [18] proposed *speculative lock-elision* (SLE), the automatic replacement of locks by optimistic hardware transactions, with the hope that the transactions will not abort due to contention, and not fail to execute due to system calls within the transaction. The SLE approach, which is set to appear in Intel's Haswell processors in 2013 [23], promises great performance benefits for read-write locks when there are low levels of write contention, because it will allow extensive concurrent reading while writing. It will of course also allow write-write parallelism that does not appear in locks. However, if transactions do fail, SLE defaults to using the original lock which has no write-read parallelism.

A few years ago, Roy, Hand, and Harris [4] proposed an all software implementation of SLE, in which transactions are executed speculatively in software, and when they fail, or if they cannot be executed due to system calls, the system defaults to the original lock. In order to synchronize correctly and get privatization, their system uses Safe(..) instrumentation for objects and a special signaling mechanism between the threads that is implemented inside the kernel. In short, speculative lock-elision is complex and requires OS patches or hardware support because one has to deal with the possible failure of the speculative calls.

This paper introduces *pessimistic software lock-elision* (PLE), a new technique for *non-speculative* replacement of read-write locks by software transactional code. At the core of our PLE algorithm is a novel design of a fully pessimistic STM system, one in which each and every transaction, whether reading or writing, is executed once and never aborts. The fact that transactions are pessimistic means that one can simply replace the locks by transactions without the need, as in SLE [18, 4], to ever revert to the original lock based code. In particular, PLE allows read-write concurrency even for contended code and even if the code includes system calls. It provides only limited write-write concurrency, but then again, read-write locks offer none.

All past STM algorithms (see [21]), including the TinySTM algorithm of Felber, Fetzer, and Reigel [17] and the TL2 STM of Dice, Shalev, and Shavit [9], are optimistic or partially optimistic: some transactions can run into inconsistencies and be forced to abort and retry. Welc et al. [5] introduced the notion of irrevocable transactions. Their system was the first to answer the need to execute systems calls within transactions, but did not relieve the programmer from having to plan and be aware of which operations to run within the specialized pessimistic transaction. Perelman et al. [7] showed a partially pessimistic STM that can support read-only transactions by keeping multiple versions of the transactions' view during its execution. Attiya and Hillel [10] presented a partially pessimistic STM that provides read-only transactions without multiple versions. However, their solution requires acquiring a read-lock for every location being read.

Our new fully pessimistic STM design is an encounter-time variation of our earlier commit-time pessimistic STM [3]. Our algorithm executes write transactions sequentially in a manner similar to [5], yet allows concurrent wait-free read-only transactions without using read-locks or multiple versions as in [10, 7]. We do so by using a TL2/LSA [9, 22] style time-stamping scheme (we can reduce the time-stamp to two bits) together with a new variation on the quiescence array mechanism of Matveev and Shavit [2]. The almost sequential execution of the pessimistic write transactions is a drawback relative to standard TL2, but also has some interesting performance advantages. The most important one is that our STM transactions do not acquire or release locks using relatively expensive CAS operations. Moreover, one does not need read-location logging and revalidation or any bookkeeping for rollback in the case of aborts. Our use of the Matveev and Shavit quiescence mechanism is a variation on the mechanism, which was originally used to provide privatization of transactions, in order to allow write transactions to track concurrent read-only transactions with little overhead. A side benefit of this mechanism is that our new fully pessimistic STM also provides implicit privatization with very little overhead (achieving implicit privatization efficiently in an STM is not an easy task and has been the subject of a flurry of recent research [2, 6, 13–15, 19, 20]).

Though our pessimistic and privatizing STM does not provide the same performance as the optimistic non-privatizing state-of-the-art TL2 algorithm, its performance is comparable in many cases we tested. In particular, this is true when there is useful non-transactional work between transactional calls. Our new pessimistic algorithm is encounter-time, which means locations are updated as they are encountered. Our benchmarks show this improves on our prior commit-time updating approach [3] both in performance and in its suitability to handling system calls within lock code. Most importantly, our new pessimistic STM offers a significant improvement over what, to the best of our knowledge, is the state-of-the-art read-write lock: across the concurrency scale and on NUMA machines, it delivers up to 5 times the lock's throughput. The parallelism PLE offers therefore more than compensates for the overheads introduced by its instrumentation.

Finally, we show how PLE fits naturally with future hardware lock-elision and transactional memory support. We explain how to seamlessly integrate PLE into Intel's hardware lock-elision (HLE) or its restricted transactional memory (RTM) [23] mechanisms, scheduled to appear in processors in 2013. In these mechanisms, transactions cannot execute if they include system calls, and they can fail if there is read-write contention on memory locations. The idea is to execute lock-code transactionally in hardware, and use PLE as the default mechanism to be executed in software if the hardware fails: in other words, elide to the better performing PLE instead of the original read-write lock. Moreover, as we explain, PLE itself can run concurrently with hardware transactions, allowing the user the benefit from both worlds: execute locks with fast hardware transactions in the fast path, or with fast software transactions in the slow path.

2 A Pessimistic Lock-Elision System

We begin by describing the new pessimistic STM algorithm at the core of our system. We will then explain how it can be used to provide non-speculative lock-elision in today's systems that do not have HTM support, and how in the future, one will be able to provide it in systems with HTM support.

2.1 Designing a Pessimistic STM

A typical transaction must read and write multiple locations. Its *read-set* and *write-set* are the sets of locations respectively read and written during its execution. If a transaction involves only reads, we call it a *read transaction*, and otherwise it is a *write transaction*. The transactional writes may be delayed until the commit phase, making the STM *commit-time* style, or may be performed directly to the memory, making the STM *encounter-time*. This paper presents a new encounter-time fully-pessimistic STM implementation that is based on our previous commit-time fully-pessimistic STM [3], in which we allow wait-free read transactions concurrently with a write transaction. Read transactions run wait-free and are never aborted. Write transactions use a lightweight *signaling mechanism* (instead of a mutex lock) to run one after the other, where a new write transaction starts when the previous one begins its commit phase; this allows the execution of one write transaction to be concurrent with the commit phase of the previous write transaction, which we show improves performance. To ensure that a read transaction sees a snapshot view of memory, each write transaction logs the previous value of the address, and at the beginning of the commit phase a write-transaction waits until all the read transactions that have started before or during its execution phase (that does not include the commit phase) have finished. To implement the synchronization between the write and read transactions we use a variant of the quiescence array mechanism of Matveev and Shavit [2] (which in turn is based on epoch mechanisms such as RCU [8]). Read transactions are made wait-free: locations being updated by a concurrent write transaction (there is only one such transaction at a time) are read from a logged value, and otherwise are read directly from memory. In addition, as a side effect, the quiescence operation provides us with an implicit privatization, which is critical for preserving the read-write lock semantics of the program when replacing the locks with transactions.

Section 2.2 presents the global variables and structures, and defines the API functions of read and write transactions. To simplify the presentation, we first consider the case of only one write transaction executing at a time with possible concurrent read transactions. Section 2.3 presents this implementation, and presents the write transaction commit that allows concurrent read transactions to complete without aborts in a wait-free manner. Next, in Section 2.4, we consider the multiple writers case, where we present a *signaling mechanism* between the write transactions that we found to be more efficient than using a simple mutex, allowing concurrency between the current write transaction's commit and the next write transaction's execution.

2.2 Global Structures

Our solution uses a version-number-based consistency mechanism in the style of the TL2 algorithm of Dice, Shalev, and Shavit [9]. The range of shared memory is divided into stripes, each with an associated local version-number (similar to [9, 17, 1]), initialized to 0.

We use a shared global version number (as introduced by [9, 22]). The global version and stripe versions are 64bit unsigned integers. Every transaction reads the global version upon start, and determines each location's validity by checking its associated stripe's version number relative to the global version.

Our *quiescence mechanism* uses a global *activity array*, that has an entry for every thread in the system. The association between the threads and the activity array entries is explained in detail later. For now assume that *N* threads have ids *[0..N-1]*, and thread *K* is associated with the entry *activity_array[K]*. We later show how to reduce the array size to be a function of the number of hardware cores. The entry for a specific thread holds the *context* of the current transaction being executed by this thread. It includes the transaction's local variables, and shared variables; the ones accessed by other threads.

Figure 1 depicts the algorithm's global variables and structures. They include the *stripe_version_array* that has a version number per stripe, the global version number, and the activity array that holds a *context* for every thread in the system. In addition, Figure 1 shows the API of read and write transactions. Every API function gets *ctx* as a first parameter the thread's context, and the thread's associated *activity_array* entry references the same context.

Every transaction's *context* has a *tx_version* variable that is used to hold the result of sampling the global version number. The *tx_version*'s initial value is the maximum 64bit value. When a transaction starts, the *tx_version* is initialized to the current global version value, and when it finishes, it is set back to the maximum 64bit value.

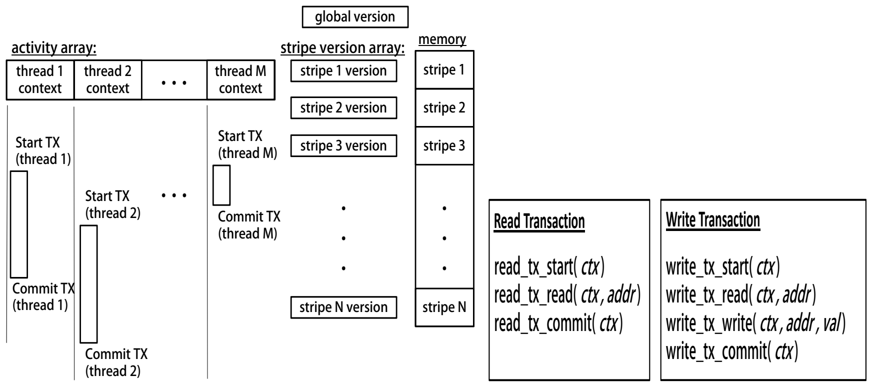


Fig. 1. The algorithm's variables and structures, and the API of a read and write transactions

2.3 The Core Algorithm

During the write transaction's execution, the write operations are written directly in memory and the overwritten values are logged to the *log_buffer*. Algorithm 1 shows the write transaction's start, read, write and commit functions. A *writer_lock* is used to allow one write transaction at a time. It is acquired on start and released on commit. The write operation logs the write location's old value to the transaction's *log_buffer*, updates the location's stripe version to the next global version and writes the new value to the memory. The order of these operations is important, because the update of the location's stripe version may require the concurrent reads of this location to snoop into the *log_buffer* of this write transaction in order to obtain the most up-to-date value.

When a write transaction commits, the global version is incremented (line 24). This splits the transactions in the system into old transactions and new transactions: ones started before the global version increment and ones that started after it.

It is consistent for new transactions to read the latest values written by the writer, because they started after the global version increment step, and can be serialized as being executed after the writer. This is not the case for old transactions that may have read old values of the locations overwritten by the writer. These transactions are not allowed to read the new values and must continue to read the old values of the overwritten locations in order to preserve their consistent memory view. As a result, old transactions perform a snoop into the concurrent writer's *log_buffer* when reading an overwritten location.

Note that the *log_buffer* values must be preserved as long as there are old read transactions that may read them. Therefore, the writer executes a *quiescence* pass (line 26)

Algorithm 1. Write transaction

```
1:  function WRITE_TX_START($ctx$)
2:      mutex_acquire($writer\_lock$)
3:      $g\_writer\_id = ctx.id$
4:      $ctx.tx\_version \leftarrow global\_version$
5:      memory_fence()
6:  end function
7:
8:  function WRITE_TX_WRITE($ctx, addr, val$)
            ▷ log the old value
9:      $n \leftarrow ctx.log\_size$
10:     $ctx.log\_buffer[n].addr \leftarrow addr$
11:     $ctx.log\_buffer[n].val \leftarrow load(addr)$
12:     $ctx.log\_size \leftarrow ctx.log\_size + 1$
            ▷ update the stripe version and write the new value
13:     $s\_index \leftarrow get\_stripe\_index(addr)$
14:     $s\_ver = stripe\_version\_array$
15:     $s\_ver[s\_index] \leftarrow ctx.tx\_version + 1$
16:     $store(addr, val)$
17: end function
18:
19: function WRITE_TX_READ($ctx, addr$)
20:     $value \leftarrow load(addr)$
21:     return $value$
22: end function
23:
24: function WRITE_TX_COMMIT($ctx$)
            ▷ allow new transactions to read the new values
25:     $global\_version \leftarrow global\_version + 1$
26:     memory_fence()
            ▷ wait for the old transactions to finish
27:     Quiescence($ctx$)
            ▷ allow the next writer to proceed
28:     mutex_release($write\_lock$)
29: end function
```

that waits for the old transactions to finish. These transactions have a *tx_version* less than the new global version (created by the global version increment) because they started before this global version increment. Therefore, it would seem sufficient to scan the *activity_array* for entries having a *tx_version* less than the new global version, and spin-loop on each until this condition becomes false. But, in this way, the scan can miss a transaction, because the *tx_version* modification is implemented as a simple load of the global version and store of the loaded value to the *tx_version*. As a result, a read transaction on start, might load a global version, the concurrent commit may perform the global version increment, and then begin the *activity_array* scan, bypassing the read transaction, because it has not yet performed the store to its *tx_version*. To overcome this scenario, we introduce a special flag, called the *update_flag*. The *tx_version*, is set to this flag value before the reading of global version to the *tx_version*, indicating that a read transaction is in the middle of *tx_version* update. In this case, the writer will wait for the update to finish by spin-looping on *tx_version* until its value becomes different than the *update_flag*'s value. In Algorithm 2 we show the implementation of the quiescence mechanism using this flag, including the read transaction start and commit procedures.

Algorithm 3 shows the implementation of the read transaction's read operation. Upon a read of a location, the transaction first validates that the location has not been overwritten by a concurrent writer by testing the location's stripe version to be less than the read transaction's *tx_version* (lines 2- 9). If validation succeeds, then the location's value is returned. Otherwise the location may have been overwritten and a snoop is performed to the concurrent writer's *log_buffer* (lines 10-28). The snoop simply scans the *log_buffer* for the read location, and returns the location's value from there (note that the scan must start from a 0 index and up, because the location may be overwritten

Algorithm 2. Quiescence

```
function TX_VERSION_UPDATE($ctx$)
    $ctx.tx\_version \leftarrow update\_flag$
    memory_fence()
    $ctx.tx\_version \leftarrow global\_version$
    memory_fence()
end function
function READ_TX_START($ctx$)
    tx_version_update($ctx$)
end function

function READ_TX_COMMIT($ctx$)
    $ctx.tx\_version \leftarrow max\_64bit\_value$
end function


function QUIESCENCE($ctx$)
    for $id = 0 \rightarrow max\_threads - 1$ do
        if $id = ctx.thread\_id$ then
            continue    ▷ to next iteration - skip this id
        end if
        $cur\_ctx \leftarrow activity\_array[id]$
        while $cur\_ctx.tx\_version = update\_flag$ do
        end while    ▷ spin-loop
        while $cur\_ctx.tx\_version < global\_version$ do
        end while    ▷ spin-loop
    end for
end function
```

twice). If this location address is not found in the log, then it means it was not overwritten (the stripe version protects a memory range), and the location's value that was read before the snoop is returned. The relevant *log_buffer* is accessed through the writer's context that is identified by a global index *g_writer_id*. This index is initialized upon write transaction start.

To illustrate the synchronization between the write and read transactions, Figure 2 shows 3 stages of a concurrent execution. In stage 1, there is read of transaction 1 and write of transaction 1; both of them read the global version on start, and proceed to reading locations. The read transaction validates that the locations were not overwritten and the writer reads them directly.

In stage 2, the writer performs two writes; (addr1, val1) and (addr2, val2). For every write; (1) the old value is stored in the log buffer, (2) the stripe version is updated, and (3) the new value is written. Then, read transaction 1 tries to read (addr1) from stripe 1 and identifies that the stripe was updated. As a result, it snoops into the concurrent writer's log buffer, searching for (addr1) old value and reading it from there. The second read of (addr3) from stripe 1, also triggers the snoop procedure, but it does not find addr3 in the log buffer and the value of (addr3) is read from the memory.

In stage 3, the writer arrives at the commit point, increments the global version and begins the quiescence step; waiting for old transactions (ones started before the increment) to finish. Specifically, the quiescence waits for read transaction 1. Meanwhile, a new read transaction 2 is started, which reads the new global version. This new transaction can read the new values freely, since it is serialized after the writer. In contrast, read transaction 2 continues to snoop into the concurrent writer log buffer until it is finished, and only then the quiescence step of the writer will finish and the log buffer will be reset. The old values are no longer required because there are no remaining active old readers.

Algorithm 3. Read Operation

```
1: function READ_TX_READ($ctx, addr$)
        ▷ Try to read the memory location
2:     $s\_index \leftarrow get\_stripe\_index(addr)$
3:     $s\_ver \leftarrow stripe\_version\_array$
4:     $ver\_before \leftarrow s\_ver[s\_index]$
5:     $value \leftarrow load(addr)$
6:     $ver\_after \leftarrow s\_ver[s\_index]$
7:     if $ver\_before <= ctx.tx\_version$ and $ver\_before = ver\_after$ then
8:         return $value$
9:     end if
        ▷ The read location may had been overwritten. Snoop into the concurrent writer's log_buffer
10:    $wr\_ctx \leftarrow activity\_array[g\_writer\_id]$
11:    $log\_size \leftarrow wr\_ctx.log\_size$
12:    $is\_found \leftarrow False$
13:    $i \leftarrow 0$
14:    while $is\_found = False$ and $i < log\_size$ do
15:        $p\_buf \leftarrow wr\_ctx.log\_buffer$
16:        $cur\_addr \leftarrow p\_buf[i].addr$
17:        $cur\_val \leftarrow p\_buf[i].val$
18:        if $cur\_addr = addr$ then
19:            $is\_found \leftarrow True$
20:            $snoop\_value \leftarrow cur\_val$
21:        end if
22:        $i \leftarrow i + 1$
23:    end while
24:    if $is\_found = False$ then
25:        return $value$
26:    else
27:        return $snoop\_value$
28:    end if
29: end function
```

2.4 The Signaling Mechanism for Write Transactions

In [5], the write transaction coordination is implemented using a global *writer_lock*. Every write transaction tries to acquire this global lock on start and release it upon finish. We have found that these lock acquire and release sequences can cause high cache coherence traffic. To avoid this, we implement a different scheme using a combination of a *writer_lock* and a simple "pass the baton" style *signaling mechanism* in the activity array.

We add to each *context* in the *activity_array* a *writer_waiting* flag. If a write transaction must wait for a concurrent writer, it sets this flag to True and spins on it until it becomes False. The concurrent writer commit scans the *activity_array* for entries having the *writer_waiting* set to True, and signals one of these entries, by changing this entry's *writer_waiting* to False. The signals must be sent in a way that avoids starvation of threads. To make the system fair we scan the activity array for an entry with a waiting writer starting from *thread_id + 1* to the array end, and from 0 to *thread_id-1*. In this way every waiting writer will be signaled after at most *max_threads* write transactions, which is proportional to the *activity_array* length.

In the common case, the write transactions will signal each other using the *writer_waiting* flags, and not by using the global lock acquire and release. That's because usually there is some degree of concurrency between the write transactions. As a result, usually during the commit of a write transaction there will be some entry in the activity array with *writer_waiting* set to True. By setting it to False, only one cache line in a specific core is invalidated, avoiding the global lock release and acquire sequences that invalidate the cache line in all of the cores.

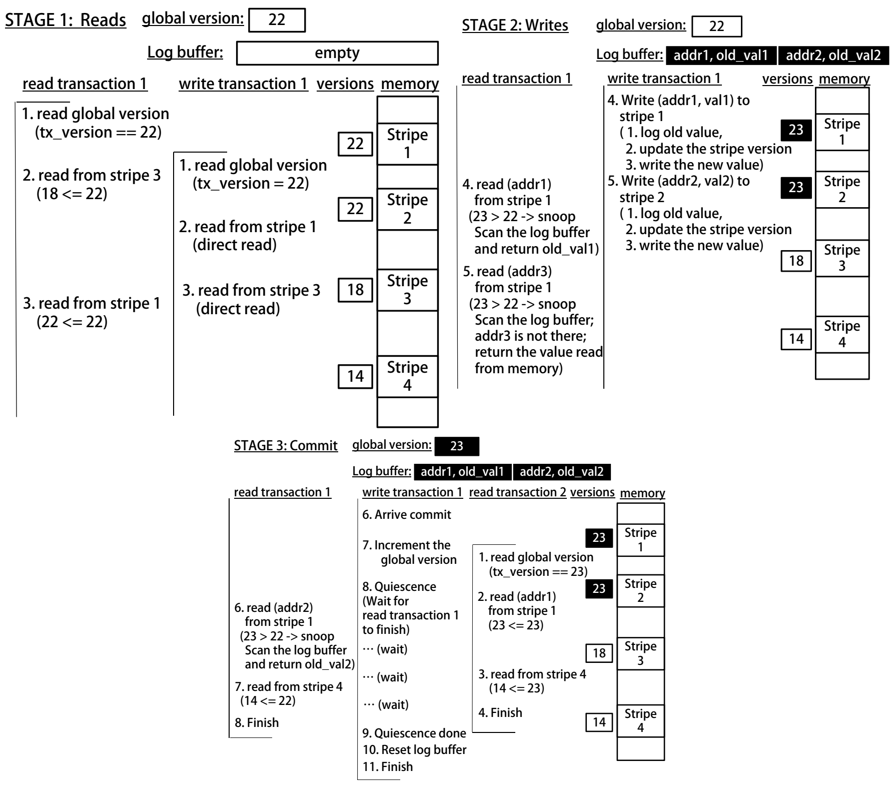

STAGE 1: Reads
global version: 22
Log buffer: empty
read transaction 1
write transaction 1
versions
memory
1. read global version (tx_version == 22)
2. read from stripe 3 (18 <= 22)
3. read from stripe 1 (22 <= 22)
1. read global version (tx_version = 22)
2. read from stripe 1 (direct read)
3. read from stripe 3 (direct read)
22 Stripe 1
22 Stripe 2
18 Stripe 3
14 Stripe 4
STAGE 2: Writes
global version: 22
Log buffer: addr1, old_val1 addr2, old_val2
4. read (addr1) from stripe 1 (23 > 22 -> snoop Scan the log buffer and return old_val1)
5. read (addr3) from stripe 1 (23 > 22 -> snoop Scan the log buffer; addr3 is not there; return the value read from memory)
4. Write (addr1, val1) to stripe 1 (1. log old value, 2. update the stripe version 3. write the new value)
5. Write (addr2, val2) to stripe 2 (1. log old value, 2. update the stripe version 3. write the new value)
23 Stripe 1
23 Stripe 2
18 Stripe 3
14 Stripe 4
STAGE 3: Commit
global version: 23
Log buffer: addr1, old_val1 addr2, old_val2
read transaction 2
6. read (addr2) from stripe 1 (23 > 22 -> snoop Scan the log buffer and return old_val2)
7. read from stripe 4 (14 <= 22)
8. Finish
6. Arrive commit
7. Increment the global version
8. Quiescence (Wait for read transaction 1 to finish)
··· (wait)
··· (wait)
··· (wait)
9. Quiescence done
10. Reset log buffer
11. Finish
1. read global version (tx_version == 23)
2. read (addr1) from stripe 1 (23 <= 23)
3. read from stripe 4 (14 <= 23)
4. Finish

Fig. 2. Three different stages of concurrent execution between read and write transactions are shown

Now the question is when to execute the signal procedure during the writer commit. The simplest way is to signal the next writer after the commit is done. In general we want to signal the next writer as soon as possible because of the writer's serial bottleneck. The earliest point for the next writer signaling is after the global version increment; immediately before the quiescence step. In this case, the snoop procedure of the read operation is complicated, because now we have the *log_buffer* of the committing writer and the *log_buffer* of the next-started writer. The snoop procedure may need to scan both of the log buffers for the read location. Therefore, we limit the number of log buffers to only two, by not allowing the next writer to signal the following writer until the current writer has finished its quiescence phase.

In summary, we have shown a pessimistic STM algorithm that allows concurrent wait-free reading while writing. We note that there are various elements algorithm that for lack of space we have not described. These include how our signaling mechanism provides better locality of reference in the critical section execution and reduced NUMA traffic by preferring to signal a transaction of a thread on the same chip to run next (up to some threshold so as to maintain fairness). They also include a mechanism to reduce the version numbers used to only two bits, allowing us to compress more of them into a single cache line in the quiescence array.

3 How to Elide Locks

We present three ways in which PLE can be used to implement lock-elision: non-speculative software-only lock elision, as a fall back (slow path) for the HLE (e.g., Intel's *hardware lock-elision* [12]), and as a fall back using optimistic hardware TM (e.g., Intel's *restricted transactional memory* RTM [12]).

3.1 Non-speculative Software Lock-Elision

To perform non-speculative elision, for every RW-Lock code section, the RW-Lock acquire and release calls are replaced with the PLE transaction start and commit calls (the read acquisition with a read transaction start and write acquisition with a write transaction start). The loads and stores are instrumented according to our above pessimistic STM algorithm with transactional read and write calls. We will denote each code section transformed into a PLE based code section as *the PLE code path* of this segment. This transformation introduces a read-write concurrency to the program that may result in two special cases:

1. *Conflicting I/O:* The concurrently executing read and write critical sections may invoke conflicting I/O requests, like a read and a write to the same file. In this case, a simple solution is to mark the conflicting I/O read critical section as a write critical section; resulting in the conflicting I/O serialization.
2. *Private Operation:* Inside the write critical section there may be a call for an operation that requires privatization (mutual exclusion) on the data it accesses. For example, a call for a free function on a shared memory. PLE provides privatization only after the commit operation and therefore these kind of operations must be moved to after the commit of the write critical section.

3.2 PLE as a Fall Back for HLE

In Intel's HLE, lock-protected code sections typically execute without locking and without interruption if they contain no system calls and if no conflicts are detected by the cache coherence protocol (there may be various other spurious reasons). If the h/w based speculation fails, it falls back to the software based locks that offer no read-write concurrency.

While Intel's HLE does not provide user specified software abort handlers, it does provide an XTEST instruction which returns true if the thread is currently executing in HLE (or RTM), and false otherwise – when an HLE or RTM transaction has been aborted. Thus, by executing XTEST after the XACQUIRE instruction (the HLE transaction start instruction), we can tell whether a fall-back to the hardware mechanism should be executed, or HLE should continue. For this to work we need to prepare at compile time a duplicate of each read-write lock protected code section where the duplicate is transformed into the corresponding PLE code path, as in the previous subsection. If the XTEST fails, then the duplicate PLE path is called.

3.3 PLE as a Fall Back for RTM

As before, each read-write lock-protected code segment is duplicated, one copy is transformed into the corresponding PLE path, as in Subsection 3.1, and the other is converted into an RTM code path as follows. Replace the acquire and release with XBEGIN and XEND, the RTM transaction start and end calls, and specify the fall-back routine (a parameter to XBEGIN) to be the matching PLE code path start. In addition, after the XBEGIN, add a read (load) instruction of a shared variable, called *is_abort*. We use *is_abort* to abort all of the hardware transactions currently executing if one of them has transitioned to PLE.

By default, each read-write lock section is first attempted as an RTM code path transaction. If it fails, a jump to the PLE pessimistic transaction start routine is performed. This routine first executes a small RTM transaction that updates the shared variable *is_abort*. This will cause all of the currently executing RTM transactions to fail. The result is a shift of the whole system to PLE. Now the PLE execution proceeds normally.

If in the RTM design, a hardware transaction is aborted when its cache line is invalidated, then we can allow execution of RTM hardware read only transactions concurrently with PLE transactions (this assumes a specific implementation of RTM which at this time we have no specfic information about [12]). This is because the PLE transactions never abort, and the cache coherence ensures that RTM hardware read-only transactions are atomic. In this case, we can avoid shifting the whole system from RTM to PLE, and shift only the write transactions.

Finally, we note that a transition from PLE back to RTM is also possible, but do not describe it here for lack of space.

4 Empirical Performance Evaluation

We empirically evaluated our algorithm on an Intel 40-way machine that includes 2 Intel Xeon E7-4870 chips on a NUMA interconnect. Each chip has 10 2.40GHz cores, each multiplex- ing 2 hardware threads (HyperThreading), and each core has private write-back L1 and L2 caches and the L3 cache is shared.

The algorithms we benchmarked are:

PLE. *Pessimistic Lock Elision:* our fully pessimistic encounter-time STM.

PTM. *Pessimistic Transactional Memory:* The commit-time variation of our fully pessimistic STM [3].

RW-Lock. An ingress-egress counter based reader-writer mutex implementation (in general, it uses a two global counters. one for read acquires and one for read releases. Writers compute the difference of these two counters to determine when there are no more readers in the system). This is state-of-the-art RW Lock implementation for Intel platform.

MCS-Lock. Michael and Scott's MCS Lock [16].

We present two standard synthetic microbenchmarks: a *Red-Black Tree* and a single location counter *(*Counter-1).

The red-black tree implementation exposes a key-value pair interface of *put*, *delete*, and *get* operations. The *put* operation installs a key-value pair, if the key is present, else

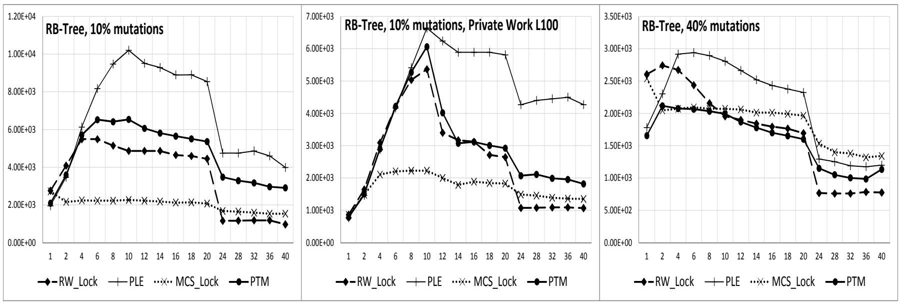


Fig. 3. Throughput of 200K sized Red-Black Tree with varying number of mutations; 10% and 40%, and varying amount of private work after the write transactions; 0 and L100 (100 dummy memory fences). The Y-axis denotes operations per second and X-axis the number of threads. Upto 10 threads every thread runs on its own core. Above 10 threads, the threads are being multiplexed, and we have 2 threads per core. From 20 threads on the second chip is being used; using the NUMA interconnect.

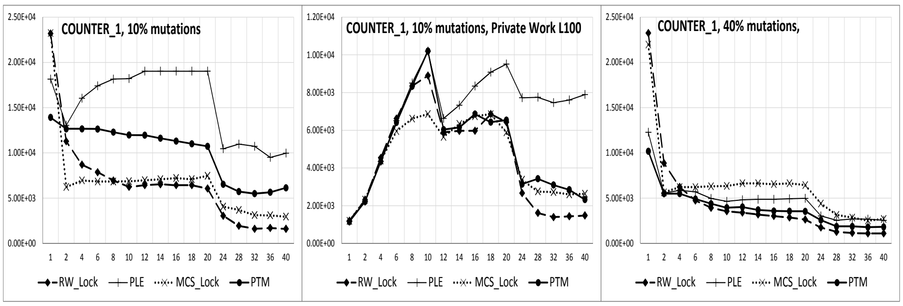


Fig. 4. Throughput of *Counter-1* benchmark with varying number of mutations; 10% and 40%

updates the key's node value. *Delete* removes the key's node, if present, and *get* returns the value associated with a key. We allow the tree to grow to a maximum of 200K elements from an initial 100K elements. We vary the fraction of mutation operations and the number of local private operations of threads after the write methods. For example, 10% mutations means we execute 5% puts and 5% deletes. We tested various rates and also a *Private Work L100* benchmark which executes 100 dummy memory fences after the write transaction.

In all the presented graphs, up to a concurrency of 10, all threads are running on separate cores on a single chip. From 11 to 20 they are being multiplexed on the 10 cores of the same chip, and from 21 to 40 they are multiplexed on the two NUMA cores of the machine.

We began by testing the benefit of allowing write concurrency in PLE and PTM. We noticed that in several benchmarks allowing write concurrency, even though it is minimal and commits are still serialized, provides a 30% performance improvement. Next, we added a priority to the *signaling mechanism* so that it will first try to signal write transactions from the same chip so as to get better locality of reference in consecutive critical section executions and avoid NUMA traffic (See [11]). We defined a constant threshold value that will limit the number of signals in the same chip, in order to avoid starvation.

As a reference point, we also compared our algorithms to the TL2 STM on the RB-Tree, despite the fact that TL2 is optimistic and non-privatizing and cannot be used to provide non-speculative lock-elision. The comparison shows that TL2 is better than PLE above about 10 threads (not included in the graphs), because in TL2 we have concurrency between the write transactions, and in the RB-Tree benchmark the number of aborts is very low. For a high number of aborts, TL2 performance degrades. Also, adding private work after the write transactions makes PLE performance similar to TL2 (upto 20 threads), because the contention is reduced.

We next ran the red-black tree benchmarks in Figure 3. Consider first the results for 10% mutations without private work (left graph) and with private work (L100 case - middle graph). For the case without private work, the MCS-Lock does not scale and the RW-Lock and PLE have similar performance until 4 threads. With more than 4 threads, PLE runs 2 times faster than the RW-Lock until 20 threads is reached. After 20 threads, we cross the boundary of one chip and start to use both of the Intel machine chips. The communication between the chips is NUMA and it is expensive, therefore, we get a performance drop in both the RW-Lock and PLE. Still, in the NUMA range (21-40 threads), PLE runs 4.5 times faster than the RW-Lock. In contrast to PLE, PTM's performance is close to that of the RW-Lock. PTM is a commit-time STM, executing more expensive write transactions. Since writers are a bottleneck, the encounter-time order of PLE makes a difference and runs faster than PTM. When there is private work, we can see that the RW-Lock, PTM, and PLE, all have a similar performance until 10 threads. Beyond 10 threads, the RW-Lock and PTM show a similar drop in the performance, while PLE runs 2 times faster than both of them until 20 threads, and 4 times faster in the NUMA range. Note, that all of the algorithms have a performance drop in the 12 threads range for the private work case. This is because the Intel machine we use starts to multiplex (use HyperThreading) from 11 to 20 threads. Above 20 threads it starts to use the second chip. We executed additional profiling analysis of the L1 cache miss rate for the benchmarks and found that the MCS-Lock has the lowest number of cache misses. Next is the RW-Lock and only then PLE. This means that a large part of PLE's performance gain is due to parallelism despite the overhead of its instrumentation and its lesser locality of reference.

In Figure 3 (right graph) we benchmark the high mutation rate of 40%. In this case, the MCS-Lock performs better than the RW-Lock above 8 threads, and PLE outperforms the RW-Lock and MCS-Lock after 4 threads and until we reach 20 threads; PLE runs 1.4 times faster in this range. In the NUMA range the MCS-Lock outperforms PLE. This is a result of a high mutation rate and lower possible concurrency between the read and write transactions. The MCS lock causes cache lines to bounce from one core to the other significantly less times.

In the *Counter-1* benchmark we model an extreme contention situation, in which every write transaction increments a shared counter and every read transaction reads this shared counter. Also, we test the case of *Private Work L100*.

Results for *Counter-1* are shown in Figure 4. For 10% mutations (write transactions) PLE is 3 times faster than RW-Lock at up to 20 threads, and 4.5 times faster in the NUMA range. For 40% mutations, the MCS-Lock outperforms both PLE and the RW-Lock because of the extreme contention on the shared counter. Again, the MCS lock has the lowest cache miss rate, though PLE's rates are not as bad as in the red-black tree benchmark. Perhaps the biased preference to signal a thread on the same node reduces bouncing of the counter cache line in PLE. For 10% mutations with private work, PLE and the RW-Mutex are similar until 8-10 threads. At 12 threads we see a performance drop because of the HyperThreading, and then we see that PLE runs 1.4 times faster until 20 threads, and 5 times faster in the NUMA range.

Acknowledgments. We thank Dima Perelman and two anonymous PODC referees for inspiring this paper by suggesting that we compare our pessimistic STMs to read write locks. This helped set us along the path of noticing that with pessimistic transactions one could actually perform straightforward non-speculative elision of read-write locks. This work was supported by the Israel Science Foundation under grant number 1386/11 and the US National Science Foundation under grant number 1217921.

References

1. Kapalka, M., Dragojevic, A., Guerraoui, R.: Stretching transactional memory. In: PLDI 2009: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 155–165. ACM, New York (2009)
2. Dice, D., Matveev, A., Shavit, N.: Implicit privatization using private transactions. In: Transact 2010, Paris, France (2010)
3. Shavit, N., Matveev, A.: Towards a fully pessimistic stm model. In: TRANSACT 2012 Workshop, New Orleans, LA, USA (2012)
4. Harris, T., Roy, A., Hand, S.: A runtime system for software lock elision. In: Proceedings of the 4th ACM European Conference on Computer Systems, EuroSys 2009, pp. 261–274. ACM, New York (2009)
5. Adl-Tabatabai, A., Welc, A., Saha, B.: Irrevocable transactions and their applications. In: SPAA 2008: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, pp. 285–296. ACM, New York (2008)
6. Attiya, H., Hillel, E.: The Cost of Privatization. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 35–49. Springer, Heidelberg (2010)
7. Keidar, I., Perelman, D., Fan, R.: On maintaining multiple versions in stm. In: Proceeding of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed computing, PODC 2010, pp. 16–25. ACM, New York (2010)
8. Desnoyers, M., Stern, A., McKenney, P., Walpole, J.: User-level implementations of read-copy update. IEEE Transactions on Parallel and Distributed Systems (2009)
9. Dice, D., Shalev, O., Shavit, N.N.: Transactional Locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
10. Attiya, H., Hillel, E.: Single-Version STMs Can Be Multi-version Permissive (Extended Abstract). In: Aguilera, M.K., Yu, H., Vaidya, N.H., Srinivasan, V., Choudhury, R.R. (eds.) ICDCN 2011. LNCS, vol. 6522, pp. 83–94. Springer, Heidelberg (2011)
11. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)
12. Intel. Intel architecture instruction set extensions programming reference – ch. 8. Document 319433-012A (2012)

13. Lev, Y., Luchangco, V., Marathe, V., Moir, M., Nussbaum, D., Olszewski, M.: Anatomy of a scalable software transactional memory. In: 4th ACM SIGPLAN Workshop on Transactional Computing, TRANSACT 2009 (2009)
14. Machens, H., Turau, V.: Avoiding Publication and Privatization Problems on Software Transactional Memory. In: Luttenberger, N., Peters, H. (eds.) 17th GI/ITG Conference on Communication in Distributed Systems (KiVS 2011), Dagstuhl, Germany. OpenAccess Series in Informatics (OASIcs), vol. 17, pp. 97–108. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2011)
15. Marathe, V., Spear, M., Scott, M.: Scalable techniques for transparent privatization in software transactional memory. In: International Conference on Parallel Processing, pp. 67–74 (2008)
16. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems 9(1), 21–65 (1991)
17. Fetzer, C., Felber, P., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 237–246. ACM, New York (2008)
18. Rajwar, R., Goodman, J.: Speculative lock elision: enabling highly concurrent multithreaded execution. In: MICRO, pp. 294–305. ACM/IEEE (2001)
19. Shpeisman, T., Menon, V., Adl-Tabatabai, A., Balensiefer, S., Grossman, D., Hudson, R., Moore, K., Saha, B.: Enforcing isolation and ordering in stm. SIGPLAN Not. 42, 78–88 (2007)
20. Spear, M., Marathe, V., Dalessandro, L., Scott, M.: Privatization techniques for software transactional memory. In: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, pp. 338–339. ACM, New York (2007)
21. Rajwar, R., Harris, T., Larus, J.: Transactional Memory, 2nd edn. Morgan and Claypool Publishers (2010)
22. Riegel, T., Felber, P., Fetzer, C.: A Lazy Snapshot Algorithm with Eager Validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 284–298. Springer, Heidelberg (2006)
23. Web. Intel tsx (2012), http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell

Asynchronous Pattern Formation by Anonymous Oblivious Mobile Robots[⋆]

Nao Fujinaga, Yukiko Yamauchi,
Shuji Kijima, and Masafumi Yamashita

Department of Informatics, Kyushu University, Japan
{fuji,yamauchi,kijima,mak}@tcslab.csce.kyushu-u.ac.jp

Abstract. We present an oblivious pattern formation algorithm for anonymous mobile robots in the asynchronous model. The robots obeying the algorithm, starting from any initial configuration I, always form a given pattern F, if I and F do not contain multiplicities and $\rho(I)$ divides $\rho(F)$, where $\rho(\cdot)$ denotes the geometric symmetricity. Our algorithm substantially outdoes an algorithm by Dieudonné et al. proposed in DISC 2010, which is dedicated to $\rho(I) = 1$. Our algorithm is best possible (as long as I and F do not contain multiplicities), since there is no algorithm that always forms F from I when $\rho(F)$ is not divisible by $\rho(I)$.

All known pattern formation algorithms are constructed from scratch. We instead use a bipartite matching algorithm (between the robots and the points in F) we proposed in OPODIS 2011 as a core subroutine, to make the description of algorithm concise and easy to understand.

1 Introduction

Autonomy is a key property required for distributed systems such as mobile sensor networks and mobile robot systems, since a huge number of inexpensive and unreliable robots (or sensor nodes) need to cooperatively behave and achieve a given task, tolerating possible failures, sometimes even without a global system initialization, particularly in the case of random deployment. A system is *self-organizing* if the robots autonomously deploy themselves to form a predefined geometric pattern from any positions, and is *self-stabilizing*, if they autonomously resume a legitimate computation, tolerating any finite number of transient failures [3]. The self-organization and the self-stabilization are two of the fundamental concepts in autonomy. In this paper, we address the problem of designing a self-stabilizing algorithm to solve the pattern formation problem for anonymous oblivious mobile robots under a fully asynchronous setting.

We model a robot by a point in a two dimensional Euclidean space. Each robot repeats a "Look-Compute-Move" cycle, where it observes the locations

[⋆] This work is supported in part by MEXT/IPSJ KAKENHI 21650002, 22300004, 23700019, 24104003 and 24650008.

of all robots in its local coordinate system (Look phase), computes a track to the next location using a given algorithm (Compute phase), and moves along the track (Move phase). Our robots have very restrictive capabilities. They are *anonymous*; they do not have identifiers, and they all execute the same algorithm. In this paper, a robot always means an anonymous robot, unless we explicitly state otherwise. They are *oblivious*; they have no memory to remember what they have observed in the past, and the algorithm computes a track in a given cycle from the observation result in this cycle. They are unaware of the global coordinate system, and their local coordinate systems may not agree, although all actions must be done in terms of their coordinate systems. Finally, their Look-Compute-Move cycles are neither instantaneous nor synchronized. We call this synchronization model the *asynchronous* (ASYNCH) model. Note that, even in the ASYNCH model, the Look phase is assumed to be instantaneous, in the sense that it returns the locations of all robots at a time.

Two stronger synchronization models are the *semi-synchronous* (SSYNCH) and the *fully-synchronous* (FSYNCH) models. In the SSYNCH model, Look-Compute-Move cycles are instantaneous, i.e., the time to execute a cycle is negligible, and in the FSYNCH model, all robots execute the i-th (instantaneous) cycle simultaneously. An essential difference between the ASYNCH and the SSYNCH models is that a robot may be observed while moving in the ASYNCH model, which situation never occur in the SSYNCH model.

The pattern formation problem in the SSYNCH and the FSYNCH models was first investigated by Suzuki and Yamashita [9]. They characterized the class of patterns F formable by non-oblivious robots when their initial configuration is I, and pointed out that *every* pattern formation algorithm for *oblivious* robots is self-stabilizing, which motivates us to emphasize oblivious robots in this paper. They [9] also showed that, in the SSYNCH model, the gathering problem for two oblivious robots is unsolvable, despite that it is trivially solvable for non-oblivious robots, which differentiates non-oblivious from oblivious robots. FSYNCH robots have stronger formation power than SSYNCH robots by definition. What is surprising is that all patterns formable by *non-oblivious FSYNCH* robots are also formable by *oblivious SSYNCH* robots, except this gathering problem for two robots [10].

The robots in [9,10] can count the number of robots at a position. Our robots do not have this *multiplicity test* capability, like those in [4,6]. As a result, for the formation to be possible, an initial configuration I and a target pattern F cannot contain multiplicities. That is, throughout the paper, we assume that I and F do not contain multiplicities.

Under this assumption, the set of patterns F formable by oblivious SSYNCH robots when their initial configuration is I can be characterized as follows [10]: Let P be a set of distinct points. The (geometric) *symmetricity* $\rho(P)$ of P is defined to be 1 if there is a point at the center $c(P)$ of the smallest enclosing circle $C(P)$ of P, and otherwise if $c(P) \notin P$, it is the number of different angles

α (between $[0, 2\pi)$) such that rotating P by α around $c(P)$ results in P. Then F is formable from I if and only if $\rho(I)$ divides $\rho(F)$. [1]

The ASYNCH model was introduced by Flocchini et al. [5]. There are only a few works about the pattern formation besides the gathering. Every pattern is formable if the robots have consistent local coordinate systems, but agreeing on one axis is not enough (without chirality) [5]. Nagamochi et al. proposed an algorithm for having all the non-oblivious robots agree on a common coordinate system, if their initial configuration is not rotation symmetry [8]. This result implies that any pattern F is formable by *non-oblivious* robots if their initial configuration I holds $\rho(I) = 1$. But their formation algorithm is not self-stabilizing. Dieudonné et al. [4] showed the same result for *oblivious* robots; their formation algorithm is now self-stabilizing. Specifically, they showed that the leader election problem is solvable starting from an initial configuration I, if and only if any pattern is formable from I. Since the leader election problem is solvable from I if and only if $\rho(I) = 1$, any pattern is formable from I if and only if $\rho(I) = 1$.

An essential difficulty in solving the formation problem is the difficulty of agreeing on a common coordinate system. For any coordinate system Z, let $Z(F)$ be the list of the coordinates $Z(p)$ (in Z) for all $p \in F$. In the pattern formation problem, a pattern F is given to the robots as $Z_0(F)$, where Z_0 is the global coordinate system (which the robots are not aware of). If $Z_i(F)$, instead of $Z_0(F)$, was given to each robot r_i, the difficulty of the problem would be substantially reduced, where Z_i is the local coordinate system of r_i; for example, the gathering problem becomes trivial. This variation of the pattern formation problem is called the *embedded pattern formation problem*, and was discussed by Fujinaga et al. [6]. They proposed an algorithm CWM for each robot r_i to find an optimum matching M_i between the robots and the points in F, and showed that all robots r_i compute the same matching $M = M_i$ and M never change no matter how each r_i approaches straight to its matched point in M. The embedded pattern formation problem is then solvable by letting each robot move toward its matched point in M.

We in this paper show that a pattern F is formable from an initial configuration I, if and only if $\rho(I)$ divides $\rho(F)$; that is, *oblivious ASYNCH* robots have exactly the same formation power as *non-oblivious FSYNCH* (and hence SSYNCH) robots. Our result includes the one in [4] as a special case $\rho(I) = 1$. Unlike the case $\rho(I) = 1$, in which a unique leader can determine a common coordinate system and control (e.g. the move order of) the robots, our robots

[1] A more formal definition of ρ is presented in Section 2. In [9,10], an (initial) configuration I is defined so that it also contains, for each robot, the description of its local coordinate system, not only its current location, and the symmetricity is defined in a similar way, taking into account the local coordinate systems. Let $\sigma(I)$ denote this symmetricity. Then [10] shows that F is formable from I if and only if $\sigma(I)$ divides $\rho(F)$. Let $P(I)$ be the locations of the robots in configuration I. For any $P(I)$, there is a configuration I' such that $P(I') = P(I)$ and $\rho(P(I)) = \sigma(I')$. Thus, as claimed here, F is formable from $P(I)$ if and only if $\rho(P(I))$ divides $\rho(F)$, in the sense that if $\rho(P(I))$ does not divide $\rho(F)$, then F may not be formable from I.

cannot take these advantages when $\rho(I) > 1$ (since for each robot there are other $\rho(I) - 1$ robots indistinguishable from it and a unique leader is not electable). This is the main contribution of this paper.

Unlike existing pattern formation algorithms, which are constructed from scratch, our algorithm uses the CWM algorithm [6], after adjustment to our problem, to make the description of our algorithm clear and concise. To invoke the CWM, some robots r_i first move to yield a "canonical" configuration so that each robot r_i can consistently "embed" F in Z_i, with a help of an imaginary coordinate system Z_i^*. Then we invoke the CWM. But some robots whose locations define Z_i^*'s cannot move in this phase; they will move to their matched points, after finishing this phase.

In the ASYNCH model, the gathering problem has been extensively investigated. (Since we assume that F does not contain multiplicities, the gathering problem is out of the scope.) See [1] for a survey of the gathering problem in the ASYNCH and related models. The gathering problem is unsolvable for two robots in the SSYNCH (and hence in the ASYNCH) model [9], and the problem for more than two robots is solvable if and only if the robots have the multiplicity test capability, in the ASYNCH [2] (and hence in the SSYNCH [9]) model.

Motivated by the fact that the gathering problem for two ASYNCH (and hence SSYNCH) robots is trivially solvable if they have consistent local coordinate systems (or reliable compasses), Izumi et al. [7] introduced unreliable compasses and characterized the solvable cases in terms of the types of compasses and their maximum deviation angles.

2 Robot Model and Pattern Formation

Robot System: Let $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ be a set of $n (\geq 3)$ ASYNCH robots in a two-dimensional Euclidean space. Note that the pattern formation is trivial if $n \leq 2$, since F does not contain a multiplicity. Each robot r_i does not have an identifier, and i is used only for the purpose of description.

A target pattern F is given to every robot r_i as a set $Z_0(F) = \{Z_0(p) : p \in F\}$, where r_i does not have access to the global coordinate system Z_0. We assume that $|Z_0(F)| = n$. In the following, as long as it is clear from the context, we identify p with $Z_0(p)$, and write, e.g., "F is given to r_i," instead of "$Z_0(F)$ is given to r_i."

We assume discrete time $0, 1, 2, \ldots$ and denote by $p_i(t)$ the location (in Z_0) of r_i at time t. $P(t) = \{p_1(t), p_2(t), \ldots, p_n(t))\}$ is a configuration at t. The robots initially occupy distinct locations, i.e., $|P(0)| = n$. $|P(t)| < n$ can hold in general, but every execution of our algorithm guarantees $|P(t)| = n$.

To define an execution of each robot r_i obeying an algorithm ψ, suppose that a Look-Compute-Move cycle of a robot r_i starts at time t and let $P(t)$ be the configuration at t. Robot r_i has a local coordinate system whose origin is always the current position of r_i. Let $Z_i(t)$ be the local coordinate system of r_i at time t. The local coordinate system $Z_i(t)$ never change in the cycle.

In the Look phase, r_i observes r_j in coordinates $Z_i(t)[p_j(t)]$ (in $Z_i(t)$), where $Z_i(t)[p_i(t)] = \mathbf{0}$ by definition.[2] The Look phase returns $Z_i(t)[P(t)]$. In the Compute phase, which starts at some time instant $t'(> t)$, from $Z_i(t)[P(t)]$ and F, algorithm ψ computes a track $X = \psi(Z_i(t)[P(t)], F)$ to the next location (in $Z_i(t)$), which *may not* be a line segment like in [4]. The Move phase, which starts at some time instant $t''(> t')$, navigates r_i to follow X, where it interprets X in $Z_i(t)$. The Move phase may finish while r_i is still on the way to the next location. We however assume that r_i always moves at least a constant distance ϵ in (Z_0), or reaches the next location if the length of X is shorter than ϵ.

We do not make any assumption when each of the phases starts. Thus the execution $P(0), P(1), \ldots$ is not uniquely determined, even for a fixed initial configuration $I = P(0)$. Rather, there are many possible executions $P(0), P(1), \ldots$, depending not only on the times when the phases start but also on the distances that the robots move in the Move phases.

Coordinate System: We assume that Z_0 and $Z_i(t)$ are right-handed coordinate systems, and let $\mathbb{T}$ be the set of all coordinate systems, which can be identified with the set of all transformations consisting of translations, rotations, and uniform scalings.

We may interpret the two-dimensional Euclidean space as the Gaussian plane, to make use of the polar form. A point (x, y) is associated with a complex number $x + jy$, which can also be represented by (r, θ) in polar form, since $x + jy = re^{j\theta}$, where $r = \sqrt{x^2 + y^2}$ is the absolute value, $\theta = \arctan(y/x)$ is the argument, and j is the imaginary unit. For $q = x + jy$ and $q = x' + jy'$, let $d(p, q) = \sqrt{|x - x'|^2 + |y - y'|^2}$ denote the Euclidean distance between p and q. A curve connecting points p and q is a set $X \subset \mathbb{C}$ such that there exists a continuous map $f : [[0, 1]] \mapsto \mathbb{C}$, $f([0, 1]) = X$, $f(0) = p$, and $f(1) = q$, where $f[D] = \{f(d) : d \in D\}$ for a set D of points.

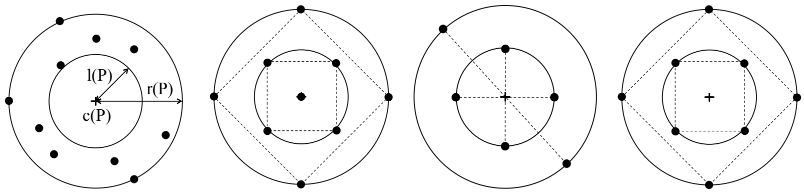
Pattern Formation: Let $\mathcal{P}_n$ be the set of all sets of n points in Z_0. For any $P, P' \in \mathcal{P}_n$, P is *similar* to P', if there exists $Z \in \mathbb{T}$ such that $Z[P] = P'$, denoted by $P \simeq P'$. Let $F, I \in \mathcal{P}_n$. We say that an algorithm ψ forms pattern F from initial configuration I, if for any execution $P(0)(= I), P(1), \ldots$, there is a time instant t_0 such that $P(t) \simeq F$ for all $t_0 < t$. A pattern F is *formable* from an initial configuration I, if there is an algorithm ψ that forms F from I.

For any $P \in \mathcal{P}_n$, let $c(P)$ be the center of the smallest enclosing circle $C(P)$. Formally, the symmetricity $\rho(P)$ of P is defined by

$$\rho(P) = \begin{cases} 1 & \text{if } c(P) \in P, \\ |\{Z \in \mathbb{T} : P = Z[P]\}| & \text{otherwise.} \end{cases}$$

We can also define $\rho(P)$ in the following way [9]: F can be divided into regular k-gons with co-center $c(P)$, and $\rho(P)$ is the maximum of such k. Here, any point is a regular 1-gon with an arbitrary center, and any pair of points $\{p, q\}$ is a regular 2-gon with its center $(p + q)/2$ (Figure 1 (b)–(d)).

[2] Note that r_i cannot compute $p_j(t)$ from $Z_i(t)[p_j(t)]$, since r_i is not aware of Z_0.



(*a*) $r(P)$ and $l(P)$ (*b*) $\rho(P) = 1$ (*c*) $\rho(P) = 2$ (*d*) $\rho(P) = 4$

Fig. 1. Symmetricity and decomposition: Black dots represent the points in P

A point on the circumference of $C(P)$ is said to be "on circle $C(P)$" and "the interior of $C(P)$" does not include the circumference. The radius of $C(P)$ is denoted by $r(P)$. We denote by $l(P)$ the radius of the largest "empty" circle $L(P)$ with its center $c(P)$, where $L(P)$ is empty in the sense that its interior does not include a point in P (Figure 1 (*a*)). Its circumference must include a point. Let $Circum(P) = C(P) \cap P$ and $Interior(P) = P \setminus Circum(P)$. The cardinality of $Circum(P)$ is denoted by $\sharp Circum(P)$.

3 Asynchronous Pattern Formation by Oblivious Robots

This section shows the following theorem.

Theorem 1. *Let $F, I \in \mathcal{P}_n$ for any $n \geq 3$. Then F is formable from an initial configuration I, if and only if $\rho(I)$ divides $\rho(F)$.*

It suffices to show that F is formable from an initial configuration I if $\rho(I)$ divides $\rho(F)$, since F is not formable from initial configuration I if $\rho(F)$ is not divisible by $\rho(I)$, even for SSYNCH robots [10]. We thus present an oblivious pattern formation algorithm ψ for $n (\geq 3)$ ASYNCH robots, provided that $\rho(I)$ divides $\rho(F)$. The case $n = 3$ is trivial as the following property states.

Property 1. Let $F, I \in \mathcal{P}_3$. Then F is formable from an initial configuration I, if $\rho(I)$ divides $\rho(F)$.

Proof. $\rho(I)$ is either 3 or 1. If $\rho(I) = 3$, the problem is trivial, since $\rho(F)$ must be 3, which implies that I and F are both equilateral triangles.

If $\rho(I) = 1$, without loss of generality, let r_1 and r_2 be the furthest pair among the three. If there are two furthest pairs, let r_1 and r_2 be the first pair in the positive orientation. Then r_3 moves straight to the corresponding point to form F. Note that the furthest pair does not change during the formation. $\square$

In the following, we therefore assume $n \geq 4$. As for F, if $c(F) \neq \mathbf{0}$ or $r(F) \neq 1$, the robots can scale and translate it so that $c(F) = \mathbf{0}$ and $r(F) = 1$ hold. So we assume $c(F) = \mathbf{0}$ and $r(F) = 1$ without loss of generality.

3.1 Outline of Algorithm

We would like to explain an outline of algorithm ψ. It uses the CWM algorithm [6] as its component. Since ψ is an oblivious algorithm, it cannot distinguish a configuration $P(t)$ from an initial configuration I. We thus use I to denote an input to ψ.

If $c(F) = \mathbf{0} \in F$, let F' be a pattern constructed from F by replacing point $\mathbf{0}$ with point $f = (0, l(F)/2)$. Algorithm ψ first forms F', and then moves a robot at f to $\mathbf{0}$ to complete F. Given an I, in ψ, if $c(I) \in I$, the robot at the center moves away by distance $l(I)/2$ from the center whenever $I \not\simeq F$, and no robots move toward $c(I)$, provided $c(F) \notin F$, which we assumed above. Obviously such modifications of F and I do not change $\rho(F)$ and $\rho(I)$. In Subsection 3.4, we will describe ψ assuming $c(F) \notin F$ and $c(I) \notin I$, for the sake of simplicity; one can reconstruct the complete ψ easily. Furthermore, we assume that $\rho(F) < n$ and F is not regular; the case $\rho(F) = n$ is treated separately (see Subsection 3.4).

Algorithm ψ consists of four phases. The aim of Phase 1 is to "embed" F. Let $\rho(F) = m_F < n$. By definition, F can be partitioned into $k_F = n/m_F$ regular m_F-gons F_i $(0 \leq i \leq k_F - 1)$ all with center $c(F) = \mathbf{0}$. Let $\rho(I) = m_I$. Similarly I can be partitioned into $k_I = n/m_I$ regular m_I-gons I_i $(0 \leq i \leq k_I - 1)$ all with center $c(F)$. A crucial observation here is that, as in [10, Sections 5.2 and 5.3], all robots that observe F even in its local coordinate system agree on an order of F_i's (resp. I_i's), such that the distance of the points in F_i (resp. I_i) from $c(F)$ (resp. $c(I)$) is no greater than that of F_{i+1} (resp. I_{i+1}), and that each robot is conscious of the group I_i it belongs to. Note that $r(F_0) = l(F) > 0$ (resp. $r(I_0) = l(I) > 0$).

Recall that we assume that m_I divides $m_F(< n)$. Phase 1 forms a regular m_F-gon H with a radius δ smaller than $l(F)/2$ and the center being $c(I)$, by sending m_F robots in $I_j (0 \leq j \leq m_F/m_I - 1)$, so that all robots can clearly distinguish the robots in H. Roughly, we achieve this goal as follows; first, the robots in I_0 move straight to $c(I)$ until the point whose distance from $c(I)$ is δ, then the robots in $I_1, I_2, \ldots, I_{m_F/m_I - 1}$ move, group by group in this order, to the adequate points.

Now, depending on the current configuration I, each robot r_i can define an imaginary coordinating system Z_i^* called the *normalized coordinate system*, taking advantage of H, as we will explain in Subsection 3.3, and they embed (roughly) F to start the CWM. Each coordinate system Z_i^* is determined by $r(I)$ and H, and is kept unchanged in Phases 2 and 3. The CWM thus enjoys the stable Z_i^*'s during the phases.

Let $F' = F \setminus H$. The aim of Phases 2 and 3 is roughly to solve the formation problem for F'. In Phase 2, the CWM is invoked by the robots besides $Circum(I)$ and H, to move their matched points.

However, the CWM may order a robot to move straight to its matched point traversing $C(H)$, which could collapse the embedding by sending robots inside H. To avoid this danger, we modify the CWM, as we will describe in Subsection 3.2.

In Phase 3, the robots in $Circum(I)$ move to their matched points, by invoking the modified CWM.

Finally in Phase 4, the robots in H resume their correct positions.

Most of the cases are described in a unified manner, but the case $\sharp Circum(F) = \sharp Circum(I) = 2$ needs a different treatment in Phase 1. We will present the whole ψ in Subsection 3.4.

3.2 Clockwise Matching on a Cylinder

Let $I = \{p_1(0), p_2(0), \ldots, p_n(0)\}$ and $F = \{f_1, f_2, \ldots, f_n\}$ be an initial configuration and a target pattern, respectively. Suppose that each robot r_i is given $Z_i(0)[F]$ (not $Z_0[F] = F$) as the description of F. Note that $Z_i(0)[I]$ is what r_i observes initially. If all r_i can agree on a bipartite matching M between I and F, that is, if each r_i can compute an element $Z_i(0)(f_{M(i)}) \in Z_i(0)[F]$ in such a way that $M = \{(p_i(0), f_{M(i)}) : 1 \leq i \leq n\}$ forms a perfect matching between I and F, and furthermore if M is optimum in the sense that the sum of the Euclidean distances between $p_i(0)$ and $f_{M(i)}$ for all r_i is minimized, then one can expect that the robots can form F by having each robot r_i move straight to $Z_i(0)(f_{M(i)})$, since these moves would not change the optimum matching M. Fujinaga et al. [6] proposed an algorithm CWM called the *clockwise matching* that produces a matching satisfying these conditions and showed that the expectation above is indeed the case. That is, let $P(0) = I, P(1), P(2), \ldots$ be any execution of the robots obeying CWM, then there is a $t \geq 0$ such that $P(t') \simeq F$ holds for any $t' \geq t$. Note that the CWM can produce the whole M (in terms of a matching between $Z_i(0)[I]$ and $Z_i(0)[F]$) not only $Z_i(0)(f_{M(i)})$.

However, our algorithm requires robots for example to avoid moving in the m_F-gon H that defines the normalized coordinate system. That is, it may force a robot to move along a curve (not only a line segment). We therefore extend the matching algorithm so that it can handle these cases.

We introduce a bijective mapping g from $\mathbb{C} \setminus \{0\}$ to $Cyl = \mathbb{R} \times S_1$, where $S_1 = [0, 2\pi)$ is the unit circle, by

$$g(re^{j\theta}) = (\log r, \theta).$$

Cyl is a side surface of a cylinder with radius 1 and an infinite height (Figure 2). Given two points $p = (h_p, \theta_p)$ and $q = (h_q, \theta_q)$, define a metric d_{Cyl} by $d_{Cyl}(p, q) = \sqrt{d_{arg}(\theta_p, \theta_q)^2 + |h_p - h_q|^2}$, where $d_{arg}(\theta, \theta') = \min\{|\theta - \theta'|, 2\pi - |\theta - \theta'|\}$. We denote by $\overline{pq}$ the (directed) line segment from p to q on Cyl. The length of $\overline{pq}$ is $d_{Cyl}(p, q)$. For $p, q \in \mathbb{C}$, we define the arc $\overset{\frown}{pq} = g^{-1}[\overline{g(p)\ g(q)}]$.

We modify the clockwise matching CWM as follows: (i) For any $p, q \in \mathbb{C}$, the distance between p, q is measured by $d_{Cyl}(g(p), g(q))$, and (ii) for any $p \in I$ and $f \in F$, edge $(p, f) \in I \times F$ now abstracts arc $\overset{\frown}{pf}$ (not line segment $\overline{pf}$).

The triangle inequality holds in Cyl, and this modification preserves the properties of the (original) clockwise matching. A notable property of $\overset{\frown}{pf} \in M$ is that it does not enter the circle with radius $\min\{r_p, r_f\}$ and the center being the

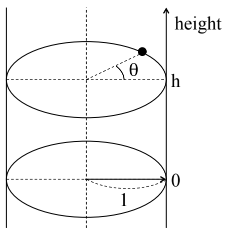


Fig. 2. Point on a side surface of a cylinder

origin, where $p = (\log r_p, \theta_p)$ and $f = (\log r_f, \theta_f)$. Indeed, if p and f are on the circumference of a circle, $\overset{\frown}{pf}$ is a part of the circumference.

3.3 Embedding a Target Pattern

Let $q \in P \in \mathcal{P}_n$, and define a coordinate system $Z(P, q)$ as described in Figure 3. Its origin is $c(P)$, the unit distance is $r(P)$, and the directions of the axes are defined by q. Observe that $Z(P, q)$ depends only on $c(P), r(P)$ and q.

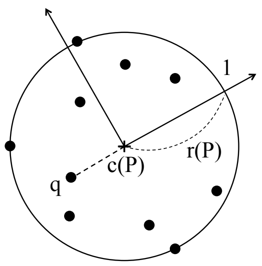
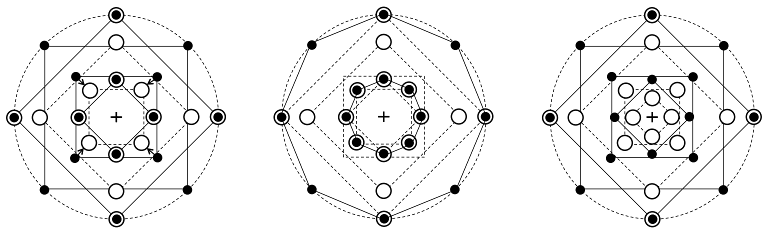


Fig. 3. $Z(P, q)$

Let F be a given pattern. Note that $c(F) \notin F$. Recall that F is partitioned into $k_F = n/m_F$ regular m_F-gons F_i. For any i and $f, f' \in F_i$, $Z(F, f)[F] = Z(F, f')[F]$ holds, by the definition of symmetricity. Define F^* by $Z(F, f)[F] = F^*$, where $f \in F_0$.

Let I be the current configuration. Recall again that I is partitioned into $k_I = n/m_I$ regular m_I-gons I_i. Since $Z(F, f)$ depends only on $c(F), r(F)$ and F_0, if I satisfies that $c(I) = c(F)$ and $F_0 = \cup_{0 \le j \le m_F/m_I - 1} I_j$, i.e., if I satisfies the conditions satisfied when Phase 1 finishes, $Z(I, q)[F] = Z(F, f) = F^*$ for any $q \in I_0$, since $q \in F_0$.

In I, each robot r_i selects (e.g., in terms of the alphabetic order over the set of the coordinates in $Z_i(t)$) a point q in I_0 and define $Z(I, q)$, which is called the *normalized coordinate system* denoted by Z_i^*. By the definition, we observe the following property, since Z_i^* is a bijective map.



(a) $\rho(I) = \rho(F) = 4$ (b) $\rho(I) = 8$ and $\rho(F) = 4$ (c) F'

Fig. 4. Construction of m_F: If we take $H = F_0$ $((a))$, $\rho(I)$ becomes larger than $\rho(F)$ $((b))$. Instead, we scale down F_0 $((c))$.

Property 2. For any two robots r_i and r_j, if $Z_i^*[F'] = Z_j^*[F''] = F^*$, then $F' = F''$.

Now, each robot r_i embeds F as the set of coordinates F^* in Z_i^*, when it invokes the CWM in Phases 2 and 3. (And during Phases 2 and 3, Z_i^* is kept unchanged.)

3.4 Algorithm ψ

We now present algorithm ψ which forms a pattern $\widetilde{F} \simeq F$ such that $c(\widetilde{F}) = c(I)$ and $r(\widetilde{F}) = r(I)$, when starting from an initial configuration I, provided that $\rho(I)$ divides $\rho(F)$. Note that $\widetilde{F} \neq F$ may hold, but to save symbols, we use F to denote $\widetilde{F}$. Indeed ψ forms F^* as $\widetilde{F}$.

As explained in Subsection 3.1, we assume (1) $n \geq 4$, (2) $c(I) \notin I$, (3) $c(F) \notin F$, (4) $c(F) = \mathbf{0}$, and (5) $r(F) = 1$. The cases in which $n = 4$ and F is regular, are exceptional cases and are explained later.

The algorithm consists of four phases, Phases 1–4. We explained that the aim of Phase 3 is to move the robots in $Circum(I)$ to the matched points, while keeping $r(I)$ unchanged, which may be impossible when $\sharp Circum(I) = 2$. Such a situation must occur if $\sharp Circum(F) = 2$ since no robot comes to $C(I)$ in Phase 2. To treat the case, a more delicate implementation of Phase 1 is requested. In the following, we first describe Phases 1–4 for I and F such that $\sharp Circum(I) \neq 2$ or $\sharp Circum(F) \neq 2$ holds, and then show Phase 1 for the case in which $\sharp Circum(I) = 2$ and $\sharp Circum(F) = 2$ hold.

Algorithm ψ for Case $\sharp Circum(I) > 2 \vee \sharp Circum(F) > 2$:
(Phase 1). The m_F-gon H is obtained by scaling down F_0 with keeping $c(F)$ unchanged so that H is placed in the interior of $L(F)$ and $L(I)$, since we cannot use F_0 as H in general for our purpose. To see this, consider Figure 4 (a), where $r(F_0) = r(F_1) = l(F)$. If the four points in $L(F_0)$ are occupied by robots (Figure 4 (b)), $\rho(I)$ becomes larger than $\rho(F)$, and violates our basic assumption that $\rho(F)$ is divisible by $\rho(I)$.

We define the radius δ of H as follows: On I, let us draw circles all with center $c(I)$, whose radii are $(r(I)l(F))/(r(F)2^k)$ for all $k = 1, 2, \ldots$. (If we place F on I in such a way that $r(I) = r(F)$, then the radii are $l(F)/2^k$.) These radii are candidates for δ. We select δ as follows: If more than $\rho(I)$ robots are on $L(I)$, δ is the largest $(r(I)l(F))/(r(F)2^k)$ such that $(r(I)l(F))/(r(F)2^k) < l(I)$, and otherwise if exactly $\rho(I)$ robots are on $L(I)$, then δ is the largest $(r(I)l(F))/(r(F)2^k)$ such that $(r(I)l(F))/(r(F)2^k) \leq l(I)$. Notice the difference between $<$ and $\leq$. Since $l(I) > 0$, δ is well defined, and $\delta < l(I)$ if there are more than $\rho(I)$ robots on $L(I)$.

Intuitively, Phase 1 forms H with radius δ, by moving robots in $I_0, I_1, \ldots, I_{m_F/m_I-1}$, group by group. Formally, we implement this process as follows: Let I be the configuration when a robot r_i gets up. It calculates δ. Let $q \in I_0$. It defines $Z_i^*(= Z(I,q))$ and embeds $F' = F \setminus F_0 + F_0/2^k$, where k is an integer such that $\delta = (r(I)l(F))/(r(F)2^k)$ holds, and $F_0/2^k = \{f/2^k : f \in F\}$ in terms of Z_i^*. The points in $I \cap F_0/2^k$ (where I is in Z_i^*) correspond to the robots already in H. For each pattern points $p \in F_0/2^k \setminus I$, we calculate the nearest robot position $p' \in I$ (ties are broken in a clockwise manner), and have each of them move along the curve $\overset{\frown}{p'p}$, if it belongs to the smallest j such that $I_j \not\subseteq F_0/2^k$.

If $q \in H$, Z_i^* does not change during Phase 1 (indeed until Phase 4), and it works as the normalized coordinate system, since it will not move until Phase 4 starts. On the other hand, if q is on the way to the circle, since $q \in I_0$, q moves toward $c(I)$, and hence Z_i^* never change by the move of q. Thus again Z_i^* can work as the stable normalized coordinate system.

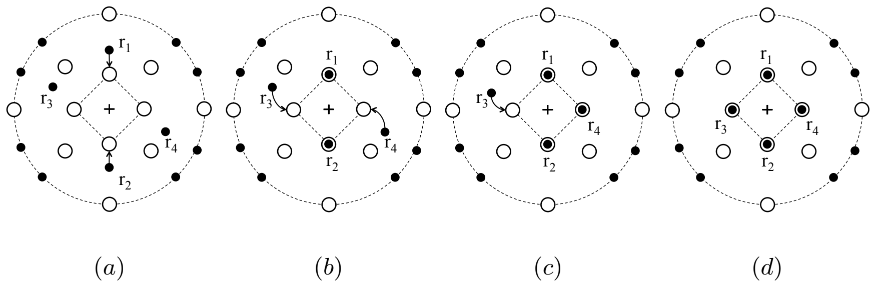


Fig. 5. An example of an execution of Phase 1. In (a), $I_0 = \{r_1, r_2\}$ and they start moving. In (b), $I_2 = \{r_3, r_4\}$ and they start moving. In (c), only r_4 reached, and r_3 starts moving. In (d), $H \subset I$ and Phase 1 finished.

(Phase 2). A robot r_i who notices that H has been formed, defines Z_i^*, embeds F', and invokes the CWM, if r_i is not on $C(I)$.

(Phase 3). A robot r_i at location $p \in I$ on $C(I)$ who notices that all robots in $Interior(I)$ have reached their destinations, calculates all the matching between the robots on $C(I)$ and the remaining points in F' as explained in Phase 1. (Note that r_i still keeps the embedding in terms of Z_i^* and this phase does not change Z_i^*.) If its matched point f is in $C(F)$, then it moves along the curve $\overset{\frown}{pf}$.

(Since both p and f are on $C(I)(= C(F))$, $\frown{pf}$ is a part of $C(I)$.) It is important to observe that there are no other robots in $\frown{pf}$, since it belongs to an optimum matching.

However, all the robots cannot gather in a half side of $C(I)$, since the size of $C(I)$ shrinks and the embedding changes. To keep $C(I)$, each robot r_i needs to check if this move does not make it critical; if this move does not change $C(P)$. If there is a point p' in $\frown{pf}$ that makes it critical, r_i moves up to the middle point of the arc $\frown{pp'}$. Since $\sharp Circum(I) > 2$ or $\sharp Circum(F) > 2$, there are at least three points on $C(I)$, and r_i eventually reaches its destination.

After all points in F_{m_F-1} have received robots, the other robots in $C(I)$, which are matched with points in $Interior(I)$, move to complete the formation for F'.

(Phase 4). Finally a robot in $p/2^k \in F_0/2^k$, when it notices that F' has been formed, move to point p, to finally form F. Because this robot heads in a straight line to p, the normalized coordinate system does not change. □

Algorithm ψ for Case $\sharp Circum(I) = 2 \wedge \sharp Circum(F) = 2$:
As explained, Phase 3 above does not correctly work when $\sharp Cicum(I) = 2$ and $\sharp Cicum(F) = 2$, since both of robots in $C(I)$ are critical. That is, they cannot change their positions in Phase 3. To remove the necessity of Phase 3, we adjust the direction of H in Phase 1, by using the locations of the robots in the last group I_{n/m_I-1}, so that they will not change during the execution. Phase 2 and Phase 4 are same as ψ, and we only explain Phase 1.

If $\rho(F) = 1$, then $\rho(I) = 1$. Let r_i be the single robot in I_0. The normalized coordinate systems of all robots are identical. Then r_i first forms H as in the Phase 1 of ψ. H is on the largest empty circle $L(I)$ centered at $c(I)$. Then r_i moves along $L(I)$ until it reaches a point p on $L(I)$, where $Z(I, p)$ embeds F in such a way that the two robots in I_{n/m_I-1} do not move during the formation.

Otherwise, $\rho(F) = 2$, robot r_i selects a point q from I_{n/m_I-1} to use $Z(I, q)$ as its imaginary coordinate system. Because $\rho(F) = 2$, the two points in I_{n/m_F-1} embeds F consistently. □

We showed the algorithm ψ. Consequently, we have the following lemma.

Lemma 1. *Let $F, I \in \mathcal{P}_n$ for any $n \geq 5$. Then F is formable from an initial configuration I, if and only if $\rho(I)$ divides $\rho(F)$.*

The remaining cases are (*i*) F is a regular n-gon, and (*ii*) $n = 4$. These cases are treated by slight modification to ψ.

Lemma 2. *Let $F, I \in \mathcal{P}_n$ for any $n \geq 4$ and $\rho(F) = n$. Then F is formable from an initial configuration I, if and only if $\rho(I)$ divides $\rho(F)$.*

Proof. If $\rho(I) = n$, the formation is completed. Otherwise, the n-gon is formed with keeping I_0 to fix the normalized coordinate systems. Let F'' be a subset of F obtained by scaling up I_0 by $r(F)/l(I)$ with keeping the center at $c(I)$. Each robot defines Z_i^* and embeds $F \setminus F''$ and moves in the same way as Phase 2 and

Phase 3 of ψ. After $F \setminus F''$ is formed, the robots in I_0 move directly to their destinations in F''. Because each robot $r_i \in I_0$ moves directly, the embedding of F does not change. □

Lemma 3. *Let $F, I \in \mathcal{P}_4$. Then F is formable from an initial configuration I, if and only if $\rho(I)$ divides $\rho(F)$.*

Proof. By Lemma 2, we can assume $\rho(F) = 1, 2$. Hence, we have two cases. If $\rho(F) = 2$, then $\sharp Circum(F/F_0) = 2$. Thus, all robots in $Circum(I)$ are critical in Phase 3 of algorithm ψ. In this case, we re-embed the target pattern by using the locations of robots in I_{n/m_F-1} as in the case $\sharp Circum(I) = \sharp Circum(F) = 2$. Then the robots in $Interior(I)$ invoke the CWM algorithm and the formation is completed.

If $\rho(F) = 1$, we have two cases. If $\sharp Circum(F) \geq 3$ or $\sharp Circum(I) \geq 3$, in Phase 3 of ψ, all robots in $Circum(I)$ are not critical and algorithm ψ forms the target pattern. Otherwise, we re-embed the target pattern as in the previous case and use the CWM algorithm to complete the formation. □

Finally, we have the following corollary and complete the proof of Theorem 1.

Corollary 1. *Let $F, I \in \mathcal{P}_n$ for any $n \geq 4$. Then F is formable from an initial configuration I, if and only if $\rho(I)$ divides $\rho(F)$.*

4 Concluding Remark

In this paper, we investigated the pattern formation problem for anonymous oblivious ASYNCH robots, and gave an algorithm ψ that forms a given pattern F from any initial configuration I, if $\rho(I)$ divides $\rho(F)$, provided that both I and F do not contain multiplicities. Since there is no pattern formation algorithm if $\rho(F)$ is not divisible by $\rho(I)$, even for non-oblivious FSYNCH robots, we conclude that oblivious ASYNCH robots have the same formation power as non-oblivious FSYNCH robots (for robots without the multiplicity test capability). We have used (a modified) clockwise matching algorithm CWM in [6] as a core procedure, which makes the description and the verification of ψ simple and compact.

Compared with the results for SSYNCH robots [10], the one here is weaker in two points; first our result does not cover patterns with multiplicities, and second it does not take into account the local coordinate systems as a tool to break a symmetric situation. Extending the result to these directions seems to be challenging, although we conjecture:

Conjecture 1. Any pattern F is formable by anonymous oblivious ASYNCH robots from an initial configuration I without multiplicities, if and only if $\sigma(I)$ divides $\rho(F)$.

Finally, not only from the view of theory, but also from the view of practice, the time necessary to form a pattern is an important performance measure. We leave an analysis of the time complexity of formation problem as another challenging open problem.

References

1. Bandettini, A., Luporini, F., Biglietta, G.: A survey on open problems for mobile robots, arXiv:1111.2259v1 (2011)
2. Cieliebak, M., Flocchini, P., Prencipe, G., Santoro, N.: Solving the Robots Gathering Problem. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 1181–1196. Springer, Heidelberg (2003)
3. Dijkstra, E.W.: Self stabilizing systems in spite of distributed control. Commu. ACM 17, 643–644 (1974)
4. Dieudonné, Y., Petit, F., Villain, V.: Leader Election Problem versus Pattern Formation Problem. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 267–281. Springer, Heidelberg (2010)
5. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. Theoretical Computer Science 407, 412–447 (2008)
6. Fujinaga, N., Ono, H., Kijima, S., Yamashita, M.: Pattern Formation through Optimum Matching by Oblivious CORDA Robots. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 1–15. Springer, Heidelberg (2010)
7. Izumi, T., Soussi, S., Katayama, Y., Inuzuka, N., Défago, X., Wada, K., Yamashita, M.: The gathering problem for two oblivious robots with unreliable compasses. SIAM J. of Comput. 41(1), 26–46 (2012)
8. Nagamochi, H., Yamashita, M., Ibaraki, T.: Distributed algorithm for cooperative controlling of anonymous mobile robots. Technical Reports of IEICE, COMP95-24, pp. 31–40 (1995) (in Japanese)
9. Suzuki, I., Yamashita, M.: Distributed anonymous mobile robots: Formation of geometric patterns. SIAM J. of Comput. 28(4), 1347–1363 (1999)
10. Yamashita, M., Suzuki, I.: Characterizing geometric patterns formable by oblivious anonymous mobile robots. Theoretical Computer Science 411, 2433–2453 (2010)

How to Gather Asynchronous Oblivious Robots on Anonymous Rings

Gianlorenzo D'Angelo[1], Gabriele Di Stefano[2], and Alfredo Navarra[3]

[1] MASCOTTE Project, INRIA/I3S(CNRS/UNSA), France
gianlorenzo.d_angelo@inria.fr
[2] Dipartimento di Ingegneria e Sceinze dell'Informazione e Matematica, Università degli Studi dell'Aquila, Italy
gabriele.distefano@univaq.it
[3] Dipartimento di Matematica e Informatica, Università degli Studi di Perugia, Italy
alfredo.navarra@unipg.it

Abstract. A set of robots arbitrarily placed on different nodes of an anonymous ring have to meet at one common node and remain in there. This problem is known in the literature as the *gathering*. Anonymous and oblivious robots operate in Look-Compute-Move cycles; in one cycle, a robot takes a snapshot of the current configuration (Look), decides whether to stay idle or to move to one of its neighbors (Compute), and in the latter case makes the computed move instantaneously (Move). Cycles are asynchronous among robots. Moreover, each robot is empowered by the so called *multiplicity detection* capability, that is, it is able to detect during its Look operation whether a node is empty, or occupied by one robot, or occupied by an undefined number of robots greater than one.

The described problem has been extensively studied during the last years. However, the known solutions work only for specific initial configurations and leave some open cases. In this paper, we provide an algorithm which solves the general problem, and is able to detect all the ungatherable configurations. It is worth noting that our new algorithm makes use of a unified and general strategy for any initial configuration, even those left open by previous works.

1 Introduction

We study one of the most fundamental problems of self-organization of mobile entities, known in the literature as the *gathering* problem (see e.g., [8,10,14] and references therein). In particular, we consider oblivious robots initially located at different nodes of an anonymous ring that have to gather at a common node and remain in there. Neither nodes nor links are labeled. Initially, each node of the ring is either occupied by one robot or empty. Robots operate in Look-Compute-Move cycles. In each cycle, a robot takes a snapshot of the current global configuration (Look), then, based on the perceived configuration, takes a decision to stay idle or to move to one of its adjacent nodes (Compute), and in the latter case it moves to this neighbor (Move), eventually. Cycles are performed asynchronously for each robot. This means that the time between Look, Compute, and Move operations is finite but unbounded, and it is decided by the adversary for each robot. Hence, robots may move based on significantly outdated perceptions. Moves are instantaneous, and hence during a Look operation robots are seen at nodes and not on edges. Robots are identical, execute the same deterministic algorithm and are empowered by the so-called *multiplicity detection* capability [15]. That is, a robot is able to perceive whether a node of the network is empty, occupied by a single robot or by more than one (i.e., a *multiplicity* occurs), but not the exact number. Without multiplicity detection the gathering has been shown to be impossible on rings [20].

Related Work. The problem of let meet mobile entities on graphs [2,11,20] or open spaces [5,10,22] has been extensively studied in the last decades. When only two robots are involved, the problem is referred to as the *rendezvous* problem [1,4,6,11,23]. Under the Look-Compute-Move model, many problems have been addressed, like the *graph exploration* and the *perpetual graph exploration* [3,12,13], while the rendezvous problem has been proved to be unsolvable on rings [20].

Concerning the gathering, different types of robot disposals on rings (configurations) have required different approaches. In particular, periodicity and symmetry arguments have been exploited. A configuration is called *periodic* if it is invariable under non-trivial (i.e., non-complete) rotation. A configuration is called *symmetric* if the ring has a geometrical *axis of symmetry*, that reflects single robots into single robots, multiplicities into multiplicities, and empty nodes into empty nodes. A symmetric configuration with an axis of symmetry has an *edge-edge symmetry* if the axis goes through two edges; it has a *node-edge symmetry* if the axis goes through one node and one edge; it has a *node-node symmetry* if the axis goes through two nodes; it has a *robot-on-axis symmetry* if there is at least one node on the axis of symmetry occupied by a robot. In [20], it is proved that the gathering is not solvable for periodic configurations, for those with edge-edge symmetry, and if the multiplicity detection capability is removed. Then all configurations with an odd number of robots, and all the asymmetric configurations with an even number of robots have been solved by different algorithms. In [19], the attention has been devoted to the symmetric cases with an even number of robots, and the problem was solved when the number of robots is greater than 18. These left open the gatherable symmetric cases of an even number of robots between 4 and 18. Most of the cases with 4 robots have been solved in [21]. The remaining ones, referred to as the set $SP4$, are symmetric configurations of type node-edge with 4 robots and the odd interval cut by the axis bigger than the even one. They are ungatherable, in general, as outlined in [19] for configurations of 4 robots on a five nodes ring. Actually, specific configurations in $SP4$ could be gatherable but requiring suitable strategies difficult to be generalized.

Finally, the case of 6 robots with an initial axis of symmetry of type node-edge, or node-node has been solved in [8]. Besides the cases left open, a unified algorithm that handles all the above cases is also missing.

Other interesting gathering results on rings concern the case of the so called *local weak* multiplicity detection. That is, a robot is able to perceive the multiplicity only if it is part of it. On this respect, our assumption in the rest of the paper concerns the *global weak* multiplicity detection. Whereas, the *strong* version would provide the exact number of robots on a node.

Using the local weak assumption, not all the cases has been addressed so far. In [16], it has been proposed an algorithm for aperiodic and asymmetric configurations with the number of robots k strictly smaller than $\lfloor \frac{n}{2} \rfloor$, with n being the number of nodes composing the ring. In [17], the case where k is odd and strictly smaller than $n-3$ has been solved. In [18], an algorithm for the case where n is odd, k is even, and $10 \leq k \leq n-5$ is provided. The remaining cases are still open and a unified algorithm like the one we are proposing here for the global weak assumption is not known.

Without any multiplicity detection, in [7] the grid topology has been exhaustively studied.

Our Results. In this paper, we present a new distributed algorithm for solving all the gatherable configurations (but those potentially in $SP4$) by using the (global weak) multiplicity detection. Our technique introduces a new approach and for some special cases makes use of previous ones. In particular, existing algorithms are used as subroutines for solving the basic gatherable cases with 4 or 6 robots from [21] and [8], respectively. Also, we exploit:

Property 1. [20] Let C be a symmetric configuration with an odd number of robots, without multiplicities. Let C' be the configuration resulting from C by moving the unique robot on the axis to any of its adjacent nodes. Then C' is either asymmetric or still symmetric but aperiodic. Moreover, by repeating this procedure a finite number of times, eventually the configuration becomes asymmetric (with possibly one multiplicity).

For all the other gatherable configurations, we design a new approach that has been suitably unified with the used subroutines. Our result answers to the posed conjectures concerning the gathering, hence closing all the cases left open, and providing a general approach that can be applied to all the initial configurations. The main result of this paper can be stated as follows.

Theorem 1. *There exists a distributed algorithm for gathering $k > 2$ robots on a ring, provided that the composed configuration does not belong to the set $SP4$, it is aperiodic, it does not admit an edge-edge axis of symmetry. The algorithm also allows robots to recognize whether a configuration is ungatherable.*

2 Definitions and Preliminaries

We consider an n-nodes anonymous ring without orientation. Initially, k nodes of the ring are occupied by k robots. During a Look operation, a robot perceives the relative locations on the ring of multiplicities and single robots. The current configuration of the system can be described in terms of the view of a robot r. We denote a configuration seen by r as a tuple $Q(r) = (q_0, q_1, \ldots, q_j)$, $j \leq k-1$, that represents the sequence of the numbers of free consecutive nodes broken up

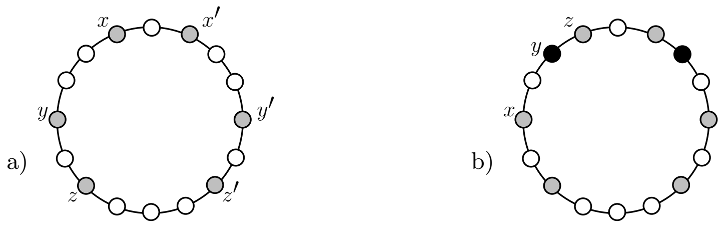


Fig. 1. a) The intervals between robots y, z and y', z' are the supermins, while the supermin configuration view is $(1, 2, 1, 2, 1, 3)$. b) Black nodes represent multiplicities.

by robots when traversing the ring in one direction, starting from r. Abusing the notation, for any $0 \leq i \leq j$, we refer by q_i not only to the length of the i-th interval but also to the interval itself. Unless differently specified, we refer to $Q(r)$ as the lexicographical minimum view among the two possibilities. For instance, in the configuration of Fig. 1a, we have that $Q(x) = (1, 2, 1, 3, 1, 2)$. A multiplicity is represented as $q_i = -1$ for some $0 \leq i \leq j$, regardless the number of robots in the multiplicity. For instance, in the configuration of Fig. 1b, $Q(x) = (1, -1, 0, 1, 0, -1, 1, 1, 3, 1)$. Given a generic configuration $C = (q_0, q_1, \ldots, q_j)$, let $\overline{C} = (q_0, q_j, q_{j-1}, \ldots, q_1)$, and let C_i be the configuration obtained by reading C starting from q_i, that is $C_i = (q_i, q_{(i+1) \bmod j+1}, \ldots, q_{(i+j) \bmod j+1})$. The above definitions imply:

Property 2. Given a configuration C,

i) there exists $0 < i \leq j$ such that $C = C_i$ iff C is periodic;
ii) there exists $0 \leq i \leq j$ such that $C = \overline{(C_i)}$ iff C is symmetric;
iii) C is aperiodic and symmetric iff there exists only one axis of symmetry.

The next definition represents the key feature for our algorithm since it has a twofold advantage. In fact, based on it, a robot can distinguish if the perceived configuration (during the Look phase) is gatherable and if it is one of the robots allowed to move (during the Compute phase).

Definition 1. *Given a configuration $C = (q_0, q_1, \ldots, q_j)$ such that $q_i \geq 0$, for each $0 \leq i \leq j$, the view defined as $C^{SM} = \min\{C_i,\ \overline{(C_i)},\ \mid 0 \leq i \leq j\}$ is called the* supermin configuration view*. An interval is called* supermin *if it belongs to the set $I_C = \{q_i \mid C_i = C^{SM}$ or $\overline{(C_i)} = C^{SM}, 0 \leq i \leq j\}$.*

The next lemma, based on Definition 1, is exploited to detect possible symmetry or periodicity features of a configuration:

Lemma 1. *Given a configuration $C = (q_0, q_1, \ldots, q_j)$ with $q_i \geq 0$, $0 \leq i \leq j$:*
1. *$|I_C| = 1$ if and only if C is either asymmetric and aperiodic or it admits only one axis of symmetry passing through the supermin;*
2. *$|I_C| = 2$ if and only if C is either aperiodic and symmetric with the axis not passing through any supermin or it is periodic with period $\frac{n}{2}$;*
3. *$|I_C| > 2$ if and only if C is periodic, with period at most $\frac{n}{3}$.*

Proof. 1.$\Rightarrow$) If $|I_C| = 1$, then if C is symmetric, there exists at least an axis of symmetry. This axis must pass through the supermin, as otherwise there exists another interval of the same size of supermin to which the supermin is reflected with respect to the axis. However, the same should hold for every neighboring interval of the supermin and so forth. Since by hypothesis, supermin is unique, there must exist at least two intervals of different sizes that are reflected by the supposed symmetry, and hence C results asymmetric.

If C is asymmetric then it must be aperiodic, as otherwise there exists $0 < i \le j$ such that $C = C_i$ and this implies more than one copy of the supermin.

1.$\Leftarrow$) If C is asymmetric and aperiodic, then $C_i \neq \overline{(C_i)}$, $C_i \neq C_\ell$ and $C_i \neq \overline{(C_\ell)}$, for each i and $\ell \neq i$ and hence must exist a unique supermin. If C admits only one axis of symmetry traversing the supermin, then there exists a unique $0 \le i \le j$ such that $C^{SM} = C_i = \overline{(C_i)}$ as otherwise Property 2 would imply the existence of other axes of symmetry, one for each supermin.

2.$\Rightarrow$) If $|I_C| = 2$ and C is asymmetric, then by Property 2, it is periodic and the period must be of $\frac{n}{2}$. If $|I_C| = 2$ and C is aperiodic and symmetric, the axis of symmetry cannot pass through both the supermins. In fact, if it does, $C^{SM} = \overline{(C^{SM})} = (C^{SM})_{j/2} = \overline{(C^{SM})_{j/2}}$ that implies $(C^{SM})_{\lfloor j/4 \rfloor} = \overline{(C^{SM})}_{\lceil j/4 \rceil}$, i.e., there exists another axis of symmetry orthogonal to the first one that reflects the supermin into the other supermin. Hence, C would be periodic.

2.$\Leftarrow$) If C is aperiodic and symmetric with the unique axis not passing through any supermin, then each supermin must be reflected by the axis to another one. Moreover, there cannot be more than 2 supermins, as by definition of supermin, these imply other axes of symmetry, i.e., by Property 2, C is periodic. If C is periodic with period $\frac{n}{2}$, then any supermin has an exact copy after $\frac{n}{2}$ intervals, and there cannot be other supermins, as otherwise the period would be smaller.

3.$\Rightarrow$) If $|I_C| > 2$, then there are at least 3 supermins, and hence C has a period of at most $\frac{n}{3}$.

3.$\Leftarrow$) If C has a period of at most $\frac{n}{3}$, then a supermin is repeated at least 3 times in C. □

2.1 A First Look to the Algorithm

The above lemma already provides useful information for a robot when it wakes up. In fact, during the Look operation, it can easily recognize if the configuration contains only 2 robots, or if it belongs to the set $SP4$, or if $|I_C| > 2$ (i.e., the configuration is periodic), or in case $|I_C| = 2$, if the configuration admits an edge-edge axis of symmetry or it is again periodic. After this check, a robot knows if the configuration is gatherable, and proceeds with its computations. Indeed, we will show in the next section that all the other configurations are gatherable. From now on, we do not consider the above ungatherable configurations.

The main strategy allows only movements that affect the supermin. In fact, if there is only one supermin, and the configuration allows its reduction, the subsequent configuration would still have only one supermin (the same as before but reduced), or a multiplicity is created. In general, such a strategy would lead asymmetric configurations or also symmetric ones with the axis passing through the supermin to create one multiplicity where the gathering will be easily finalized by collecting at turn the closest robots to the multiplicity.

For gatherable configurations with $|I_C| = 2$, our algorithm requires more phases before creating the final multiplicity where the gathering ends. In this case there are two supermins that can be reduced. If both are reduced simultaneously, then the configuration is still symmetric and gatherable. Possibly, it contains two symmetric multiplicities. In fact, this is the status that we want to reach even when only one of the two supermins is reduced. In general, the algorithm tries to preserve the original symmetry or to create a gatherable symmetric configuration from an asymmetric one. It is worth to remark that in all symmetric configurations with an even number of robots, the algorithm always allows the movement of two symmetric robots. Then it may happen that, after one move, the obtained configuration is either symmetric or it is asymmetric with a possible *pending* move. In fact, if only one robot among the two allowed to move performs its movement, it is possible that its symmetric one either has not yet started its Look phase, or it is taking more time. If there might be a pending move, then the algorithm forces it before any other decision.

In contrast, asymmetric configurations cannot produce pending moves as the algorithm allows the movement of only one robot. In fact, we reduce the unique supermin by deterministically distinguish among the two adjacent robots, until one multiplicity is created. Finally, all the other robots will join the multiplicity one-by-one. In some special cases, from asymmetric configurations at one "allowed" move from symmetry (i.e., with a possible pending move), robots must guess which move would have been realized from the symmetric configuration, and force it in order to avoid unexpected behaviors. By doing this correctly, the algorithm brings the configuration to have two symmetric multiplicities as above, eventually. From here, a new phase that collects all the other robots but two into the multiplicities starts. Still the configuration may move from symmetric configurations to asymmetric ones at one move from symmetry. Once the desired symmetric configuration with two multiplicities and two single robots is reached, a new phase starts and moves the two multiplicities to join each other. The node where the multiplicities join represents the final gathering location.

3 Gathering Algorithm

The algorithm works in 5 phases that depend on the configuration perceived by the robots, see Fig. 2. First, it starts from a configuration without multiplicities and performs phase MULTIPLICITY-CREATION whose aim is to create one multiplicity, where all the robots will eventually gather, or a symmetric configuration with two multiplicities. In the former case, phase CONVERGENCE is performed to gather all the robots into the multiplicity. In the latter case, phases COLLECT and then MULTIPLICITY-CONVERGENCE are performed in order to first collect all the robots but two into the two multiplicities and then to join the two multiplicities into a single one. After that, phase CONVERGENCE is performed. Special cases of 7 nodes and 6 robots are considered separately in phase SEVEN-NODES.

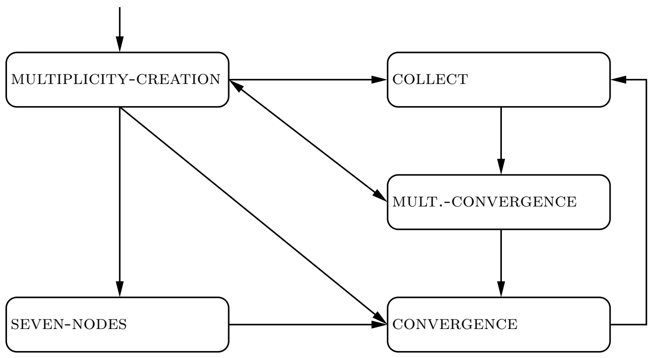


Fig. 2. Phases interchanges

Due to space constraints, we do provide the details only for the first phase MULTIPLICITY-CREATION. The formal descriptions of the other phases can be found in the full version of the paper [9].

We can show how robots interchange from one phase to another until the final gathering is achieved. In each phase we can distinguish the type of configuration and provide the algorithm to be performed by robots for each of these types. The way how a robot can identify the type of configuration will be outlined later.

3.1 Phase Multiplicity-Creation

The main idea is to reduce the supermin by enlarging the largest interval adjacent to it as follows:

Definition 2. *Let $Q(r) = (q_0, q_1, \ldots, q_j)$ be a supermin configuration view, then robot r performs the* REDUCTION *if the obtained configuration after its move is $(q_0 - 1, q_1, \ldots, q_j + 1)$.*

The pseudo-code of REDUCTION is shown above. The procedure, first checks whether the robot perceives the supermin configuration view by comparing the configuration C perceived by the robot with C^{SM}. Note that, in asymmetric configurations, the robot that perceived C^{SM} is the one among the two robots at the sides of the supermin allowed to move. In fact, the robot on the other side would perceive the configuration $\overline{C}$ and, by definition of C^{SM}, we have $C^{SM} = C < \overline{C}$, as the configuration is asymmetric. Then, the procedure moves the robot towards the supermin. In symmetric configurations, the test at line 1 returns true for both robots adjacent to the unique supermin or for the two symmetric robots that perceive C^{SM} in case that $|I_C| = 2$.

It is worth to note that, from a symmetric configuration, always two robots can perform the REDUCTION when it is possible to perform it. If only one of them does it, the obtained configuration will contain exactly one supermin. However, if the perceived configuration contains only one supermin and it is not symmetric, the robots are able to understand whether there might be a pending move to

```
Procedure: REDUCTION
Input: $C = (q_0, q_1, \dots, q_j)$
1 if $C = C^{SM}$ then move towards $q_0$;
```

re-establish the original symmetry or not. This constitutes one of the main results of the paper and it is obtained from the following lemma.

Lemma 2. *Let C be a configuration with more than 2 single robots and let C' be the one obtained from C after a* REDUCTION *performed by a single robot. If C is asymmetric then C' is at least at two moves from a symmetric configuration; if C is symmetric then C' is at least at two moves from any other symmetric configuration with an axis of symmetry different from that of C.*

Proof. By Lemma 1, two cases may arise: there exists only one supermin in C or the configuration is symmetric and contains exactly two supermins.

We now show that in the case that there exists only one supermin in C, then C' is at least two moves from any symmetric configuration with the axis different from that passing through the supermin. By Lemma 1, it is enough to show that C' requires more than one move to create another supermin different from that of C. Let us consider the supermin configuration view $C^{SM} = (q_0, q_1, \dots, q_j)$. For the sake of simplicity, let us assume that, for each $i = 1, 2, \dots, j$, $(C^{SM})_i < \overline{(C^{SM})_i}$, the case where, for some i, $(C^{SM})_i > \overline{(C^{SM})_i}$ is similar. The case that $(C^{SM})_i = \overline{(C^{SM})_i}$ cannot occur as, otherwise, there exists an axis of symmetry passing through q_i, but, by Lemma 1, as $|I_C| = 1$, the possible axis of symmetry can only pass through q_0. By definition of supermin, for each $(C^{SM})_i$, $i = 1, 2, \dots, j$, there exists $k_i \in \{0, 1, \dots, j\}$ such that: $q_\ell = q_{(i+\ell) \bmod j+1}$, for each $\ell < k_i$; and $q_{k_i} < q_{(i+k_i) \bmod j+1}$. Note that $(i + k_i) \bmod j + 1 \neq 0$ as otherwise it contradicts the hypothesis of minimality of q_0. Moreover, $k_i \neq j$ as otherwise $\sum_{\ell=0}^{j} q_\ell = \sum_{\ell=0}^{k_i} q_\ell < \sum_{\ell=0}^{k_i} q_{(i+\ell) \bmod j+1} = \sum_{\ell=0}^{j} q_{(i+\ell) \bmod j+1}$, that is a contradiction. From C', the supermin configuration view is $C'^{SM} = (q'_0, q'_1, \dots, q'_j) = (q_0 - 1, q_1, \dots, q_j + 1)$ and we have that, for each $i = 1, 2, \dots, j$, two cases may arise: if $k_i > 0$, then $q'_0 = q_0 - 1 < q_i = q'_i$ and $q'_{k_i} = q_{k_i} < q_{(i+k_i) \bmod j+1} = q'_{(i+k_i) \bmod j+1}$; if $k_i = 0$, then $q'_0 = q_0 - 1 < q_i - 1 = q'_i - 1$. In any case, C'^{SM} differs from $(C'^{SM})_i$ by two units. It follows that C' is at least two moves from any symmetric configuration with the axis different from that passing through the supermin. In fact, in order to obtain another axis of symmetry by performing only one move on C', $(C'^{SM})_i$ has to differ from C'^{SM} by at most one unit. This is enough to show the statement for the case of symmetric configurations with exactly one supermin. Regarding the asymmetric case, it remains to show that C' is at least two moves from any symmetric configuration with the axis passing through the supermin. In an asymmetric configuration $C^{SM} = (q_0, q_1, \dots, q_j)$ there exists a q_k, $1 \leq k \leq \frac{j}{2}$, such that $q_\ell = q_{(j+1-\ell) \bmod j+1}$, for each $\ell < k$, and $q_k < q_{j+1-k}$. From C', the supermin configuration view is $C'^{SM} = (q'_0, q'_1, \dots, q'_j) = (q_0 - 1, q_1, \dots, q_j + 1)$

```
Function: SYMMETRIC
Input   : $C = (q_0, q_1, \ldots, q_j)$
Output: true if $C$ is symmetric, false otherwise
1 for $i = 0, 1, \ldots, j$ do
2     if $C = \overline{C_i}$ then return true;
3 return false;
```

and two cases may arise: if $k > 1$, then $q'_1 = q_1 = q_j < q_j + 1 = q'_j$ and $q'_k = q_k < q_{j-1-k} = q'_{j-1-k}$; if $k = 1$, then $q'_1 = q_1 < q_j = q'_j - 1$. It follows that C' is at least two moves from any symmetric configuration with the axis passing through the supermin.

Regarding the case of symmetric configurations with exactly 2 supermins, we use similar arguments as above. Let us consider the supermin configuration view $C^{SM} = (q_0, q_1, \ldots, q_j)$ and let us assume that h is the index such that $C^{SM} = \overline{(C^{SM})_h}$. By definition, for each $(C^{SM})_i$, $i \in \{1, 2, \ldots, j\} \setminus \{h\}$, there exists $k_i \in \{0, 1, \ldots, j\}$ such that: $q_\ell = q_{(i+\ell) \bmod j+1}$, for each $\ell < k_i$, and $q_{k_i} < q_{(i+k_i) \bmod j+1}$. As above we are assuming that $(C^{SM})_i < \overline{(C^{SM})_i}$ and we can show that $k_i \neq j$, $k_i \neq (j + h) \bmod j + 1$, $(k_i + i) \bmod j + 1 \neq 0$, and $(k_i + i) \bmod j + 1 \neq h$. From C', the supermin configuration view is $C'^{SM} = (q'_0, q'_1, \ldots, q'_j) = (q_0 - 1, q_1, \ldots, q_j + 1)$ we have that, for each $i \in \{1, 2, \ldots, j\} \setminus \{h\}$ two cases may arise: if $k_i > 0$, then $q'_0 = q_0 - 1 < q_i = q'_i$ and $q'_{k_i} = q_{k_i} < q_{(i+k_i) \bmod j+1} = q'_{(i+k_i) \bmod j+1}$; if $k_i = 0$, then $q'_0 = q_0 - 1 < q_i - 1 = q'_i - 1$. In any case, C'^{SM} differs from $(C'^{SM})_i$ by two units. Similar arguments to the ones used for the asymmetric case can show that C' is at least two moves from any symmetric configuration with the axis passing through the supermin. □

It follows that we can derive C from C' by enlarging the supermin of C'. This equals to reduce the largest adjacent interval (i.e., by performing the REDUCTION backwards) hence deducing the possible original axis of symmetry and then performing the possible pending REDUCTION. Before providing the designed procedures, we need the following definition:

Definition 3. *Given a configuration $C = (q_0, q_1, \ldots, q_j)$, the view defined as $C^{SSM} = \min\{C_i, \overline{(C_i)} \mid C_i \neq C^{SM}$ and $\overline{(C_i)} \neq C^{SM}, 0 \leq i \leq j\}$ is called the* second supermin configuration view.

As shown above, Procedure SYMMETRIC checks whether a configuration C is symmetric by exploiting Property 2. Procedure CHECK_REDUCTION checks whether an asymmetric configuration C has been obtained from some symmetric configuration $\hat{C}$ by performing REDUCTION. Procedure PENDING_REDUCTION performs the pending REDUCTION.

At line 1, Procedure CHECK_REDUCTION looks for the index k such that q_k is the supermin, as it is the only candidate for being the interval that has been reduced by a possible REDUCTION. Then, at lines 2–4, it computes the

```
Function: CHECK_REDUCTION
Input  : $C = (q_0, q_1, \ldots, q_j)$
Output: (true, $\hat{C}$) if $C$ is obtained from $\hat{C}$ by performing REDUCTION, (false, $\emptyset$)
        if $C$ has not been obtained by performing REDUCTION
1 Let $k$ such that $C_k = C^{SM}$ or $\overline{C}_k = C^{SM}$;
2 if $q_{(k-1) \bmod j+1} > q_{(k+1) \bmod j+1}$ then
  $\hat{C} := (q_0, q_1, \ldots, q_{(k-1) \bmod j+1} - 1, q_k + 1, \ldots, q_j)$;
3 else
4     if $q_{(k-1) \bmod j+1} < q_{(k+1) \bmod j+1}$ then
      $\hat{C} := (q_0, q_1, \ldots, q_k + 1, q_{(k+1) \bmod j+1} - 1, \ldots, q_j)$;
5     else return (false,$\emptyset$);
6 if SYMMETRIC($\hat{C}$) then return (true, $\hat{C}$);
7 return (false,$\emptyset$);
```

```
Procedure: PENDING REDUCTION
Input: $C = (q_0, q_1, \ldots, q_j)$
1 $(b, \hat{C})$ = CHECK_REDUCTION ($C$) ;
2 if $b$ and ( $\min\{\hat{C}, \overline{\hat{C}_j}\} = \hat{C}^{SM}$ or $\min\{\hat{C}, \overline{\hat{C}_j}\} = \hat{C}^{SSM}$ ) and $\min\{C, \overline{C_j}\} \neq C^{SM}$
   then move towards $q_0$;
```

configuration $\hat{C}$ before the possible REDUCTION. This is done by enlarging q_k and reducing the largest interval among $q_{(k-1) \bmod j+1}$ and $q_{(k+1) \bmod j+1}$. If $q_{(k-1) \bmod j+1} = q_{(k+1) \bmod j+1}$ or $\hat{C}$ is not symmetric, then C has not been obtained by performing a REDUCTION from a symmetric configuration. Then, the procedure returns (false, $\emptyset$). If $\hat{C}$ is symmetric, then C has been obtained by performing REDUCTION on $\hat{C}$ and hence the procedure returns (true, $\hat{C}$).

Procedure PENDING_REDUCTION uses CHECK_REDUCTION to check whether C has been obtained by performing REDUCTION on a configuration $\hat{C}$ (lines 1 and 2). At line 2 the procedure checks whether the robot is at a side of one of the two supermins of $\hat{C}$ ($\min\{\hat{C}, \overline{\hat{C}_j}\} = \hat{C}^{SM}$) and if it has not yet performed REDUCTION ($\min\{C, \overline{C_j}\} \neq C^{SM}$). In the affirmative case, the robot has to move towards the supermin (line 2). The robot moves towards q_0 also if it is at the side of the second supermin of $\hat{C}$ ($\min\{\hat{C}, \overline{\hat{C}_j}\} = \hat{C}^{SSM}$). This corresponds to a move different from REDUCTION that will be explained later in this section.

In general, it is not always possible to perform REDUCTION. In fact, there are cases where it may lead to ungatherable configurations. These cases will be managed separately. However, we will show that a robot is always able to understand that there might be a pending move also for the other moves allowed by our algorithm from symmetric configurations.

When it is not possible to perform REDUCTION, we either reduce the second supermin or we perform the XN move that is defined in the following:

Definition 4. *Let C be a configuration:*

- *If C is symmetric and there are no multiplicities,* XN *corresponds to moving towards the axis the two symmetric robots closest to the axis of symmetry that are divided by at most one robot and are not adjacent to a supermin*;[1]
- *If C is symmetric and there is only one multiplicity,* XN *corresponds to moving towards the multiplicity the two symmetric robots closest to the multiplicity;*
- *If C is asymmetric and it has been possibly obtained by applying* XN *from a symmetric configuration C' (that is, from C' only one of the two robots on the above cases has moved), then* XN *on C corresponds to moving the second closest robot towards the axis/multiplicity;*
- *If C is asymmetric with a multiplicity and it cannot be obtained by applying* XN *from a symmetric configuration, then* XN *corresponds to moving the robot lexicographically closest to the multiplicity towards it.*

Each time a robot wakes up, it needs to find out which kind of configuration it is perceiving, and, if it is allowed to move, it needs to compute the right move to be performed. We need to distinguish among several types of configurations, requiring different strategies and moves. In this phase, as there are no multiplicities, a robot must distinguish among the following configurations:

W1 Symmetric configurations with an odd number of robots;
W2 Configurations with 4 robots;
W3 Configurations with 6 robots;
W4 Symmetric configurations with an even number of robots greater than 6, only 1 supermin of size 0 or with 2 supermins of size 0 divided by one interval of even size with no other intervals of size 0;
W5 Symmetric configurations with an even number of robots greater than 6, only 1 supermin of size 0 or with 2 supermins of size 0 divided by one interval of even size, and other intervals of size 0;
W6 Asymmetric configurations with an even number of robots greater than 6 and:
  a) only one interval of size 0, and it is in between two intervals of equal size;
  b) only two intervals of size 0, with only one in between two intervals of equal size;
  c) only two intervals of size 0, with one even interval in between;
  d) only three intervals of size 0, with only two of them separated by an even interval;
  e) only three consecutive intervals of size 0;
  f) only four intervals of size 0, with only three of them consecutive;
W7 Remaining gatherable configurations.

From configurations in W1, only the robot on the axis can move in one of the two directions, arbitrarily. After this move either the configuration contains one

[1] By Lemma 1, in gatherable configurations, the axis of symmetry cannot pass through two supermins hence there are always two robots allowed to move.

multiplicity or it belongs to W1 or W7. Configurations in W7 will be described later in this section and the configurations with multiplicities will be described within the other phases. Regarding configurations in W1, from Property 1, we know that the number of times that the obtained configuration can belong again to W1 after this move is bounded.

When the configuration is in W2 or W3, a modified version of algorithms in [21] and [8] are performed, respectively. In particular, both the algorithms are able to manage symmetric configurations and to check whether in an asymmetric configuration there is a possible pending move. If the configuration is not symmetric and there are no pending moves, then REDUCTION is performed. The resulting configuration is still in W2 or W3 or at least one multiplicity is created. From the correctness of algorithms in [21] and [8] and from the fact that performing REDUCTION results in reducing the supermin, it follows that eventually at least one multiplicity is created.

When the configuration is in W7 and it is symmetric, then the algorithm performs REDUCTION on two symmetric robots that leads to another symmetric configuration in W7, or to a configuration with at least one multiplicity, or to an asymmetric configuration with a pending move. In this latter case, by Lemma 2 the algorithm recognizes that the configuration is at one "allowed" move from symmetry and performs the pending move (even though it was not pending, indeed). When the configuration is asymmetric, again REDUCTION is performed. By performing the described movements, at least one multiplicity is created.

Configurations in W4–W6 correspond to the cases where REDUCTION is not allowed to be performed. In fact, if the configuration is symmetric and there is only one supermin of size 0, then REDUCTION may result in swapping the robots at the borders of the supermin, hence obtaining infinitely many times the same configuration. Similarly, if the configuration is symmetric and there are two symmetric supermins of size 0 divided by one interval of even size, then REDUCTION would produce two multiplicities divided by the interval of even size and we won't be able to join such multiplicities afterwards.

From W4, the algorithm performs XN, hence leading to configurations in W4, W6 or to configurations with one multiplicity on the axis.

From W5, the algorithm performs REDUCTION on the configuration obtained without considering the supermin (that is, it reduces the second supermin, according to Definition 3). Note that, as in this case the second supermin has size 0, we obtain at least one multiplicity.

The asymmetric configurations in W6 are either asymmetric starting configurations or are obtained from the symmetric configurations in W4 after performing XN. In this cases, the algorithm checks whether the configuration is obtained after an XN move. This is realized by moving backward the robot closest to the other pole of the axis of symmetry that is assumed to pass through: The supermin in case a); The only interval of size 0 adjacent to two intervals of equal size in case b); The even intervals mentioned in cases c) and d); the only interval of size 0 in between other two intervals of size 0 in cases e) and f). If a backwards

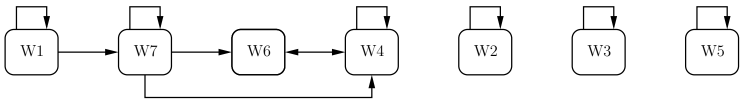


Fig. 3. Phase MULTIPLICITY-CREATION

XN produces a symmetric configuration, then the symmetric XN is performed, otherwise, REDUCTION is performed and this move creates a multiplicity.

For each configuration type, the algorithm checks whether the robot perceiving the configuration C is allowed to move and eventually, performs the move.

This phase of the algorithm is summarized in Fig. 3. The next lemma states that such a phase eventually ends with at least one multiplicity and hence one of the other phases starts.

Lemma 3. *Phase* MULTIPLICITY-CREATION *terminates with at least one multiplicity after a finite number of moves.*

Proof. From the description provided before this lemma, it follows that the graph in Fig. 3 models the execution of phase MULTIPLICITY-CREATION. We now show that all the cycles are traversed a finite number of times. This implies that eventually at least one multiplicity is created.

From Property 1, and results in [21], and [8] follows that the self-loops in W1, W2, and W3, respectively, are traversed a finite number of times.

The self-loop in W7 is traversed by performing REDUCTION or PENDING_REDUCTION. Each time such moves are performed, the supermin decreases until, after a finite number of moves, it either creates a multiplicity or leads to configurations in W4 or W6. The number of moves is at most two times the size of the initial supermin and this is obtained for symmetric configurations with the axis not passing through the supermin.

The self-loop in W4 and the cycle between W4 and W6 are traversed by performing XN. Each time this happens, the interval between the two symmetric robots closest to the axis of symmetry (excluding those adjacent to the supermin) is reduced until creating a multiplicity on the axis. The number of moves performed equals the initial size of such an interval. □

3.2 Further Notes on the Algorithm

We can show (see [9]) that all the types of configurations defined for the 5 phases are pairwise disjoint and that they cover all the possible configurations reachable by the algorithm.

Given a configuration C, in order to distinguish among the types, it is sufficient for a robot to compute simple parameters: number of nodes in the ring; number of multiplicities; number of robots (if possible) or number of occupied nodes; distances between robots and multiplicities; if C is symmetric; if C is at one move from the symmetries allowed by the algorithm.

The starting configuration can only belong to W1–W7. By Lemma 3, it follows that after a finite number of moves any other phase can be reached. Moreover, once reached a configuration with at least one multiplicity, the algorithm never goes back to configurations without multiplicities, but for a bounded number of times on some symmetric configurations with 6 robots.

The possible interactions among the phases are shown in Fig. 2, and we prove that all the possible cycles can be traversed a limited number of times, until reaching phase CONVERGENCE without leaving it anymore.

4 Conclusion

The proposed algorithm answers to the posed conjectures concerning the gathering on the studied model by providing a complete characterization for the initial configurations. The obtained result is of main interest for robot-based computing systems. In fact, it closes all the cases left open with the exception of potentially gatherable configurations in $SP4$. Our technique, mostly based on the supermin concept, may result as a new analytical approach for investigating related distributed problems.

References

1. Alpern, S.: The rendezvous search problem. SIAM J. Control Optim. 33, 673–683 (1995)
2. Bampas, E., Czyzowicz, J., Gąsieniec, L., Ilcinkas, D., Labourel, A.: Almost Optimal Asynchronous Rendezvous in Infinite Multidimensional Grids. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 297–311. Springer, Heidelberg (2010)
3. Blin, L., Milani, A., Potop-Butucaru, M., Tixeuil, S.: Exclusive Perpetual Ring Exploration without Chirality. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 312–327. Springer, Heidelberg (2010)
4. Chalopin, J., Das, S.: Rendezvous of Mobile Agents without Agreement on Local Orientation. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 515–526. Springer, Heidelberg (2010)
5. Cord-Landwehr, A., Degener, B., Fischer, M., Hüllmann, M., Kempkes, B., Klaas, A., Kling, P., Kurras, S., Märtens, M., Meyer auf der Heide, F., Raupach, C., Swierkot, K., Warner, D., Weddemann, C., Wonisch, D.: A New Approach for Analyzing Convergence Algorithms for Mobile Robots. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 650–661. Springer, Heidelberg (2011)
6. Czyzowicz, J., Labourel, A., Pelc, A.: How to meet asynchronously (almost) everywhere. In: Proc. of the 21st ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 22–30 (2010)
7. D'Angelo, G., Di Stefano, G., Klasing, R., Navarra, A.: Gathering of Robots on Anonymous Grids without Multiplicity Detection. In: Even, G., Halldórsson, M.M. (eds.) SIROCCO 2012. LNCS, vol. 7355, pp. 327–338. Springer, Heidelberg (2012)

8. D'Angelo, G., Di Stefano, G., Navarra, A.: Gathering of Six Robots on Anonymous Symmetric Rings. In: Kosowski, A., Yamashita, M. (eds.) SIROCCO 2011. LNCS, vol. 6796, pp. 174–185. Springer, Heidelberg (2011)
9. D'Angelo, G., Di Stefano, G., Navarra, A.: How to gather asynchronous oblivious robots on anonymous rings. Rapport de recherche RR-7963, INRIA (2012)
10. Degener, B., Kempkes, B., Langner, T., Meyer, F.: auf der Heide, P. Pietrzyk, and R. Wattenhofer. A tight runtime bound for synchronous gathering of autonomous robots with limited visibility. In: Proc. of the 23rd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA), pp. 139–148 (2011)
11. Dessmark, A., Fraigniaud, P., Kowalski, D., Pelc, A.: Deterministic rendezvous in graphs. Algorithmica 46, 69–96 (2006)
12. Flocchini, P., Ilcinkas, D., Pelc, A., Santoro, N.: Computing without communicating: Ring exploration by asynchronous oblivious robots. Algorithmica (to appear)
13. Flocchini, P., Ilcinkas, D., Pelc, A., Santoro, N.: Remembering without memory: Tree exploration by asynchronous oblivious robots. Theoretical Computer Science 411(14-15), 1583–1598 (2010)
14. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Gathering of asynchronous robots with limited visibility. Theor. Comput. Sci. 337, 147–168 (2005)
15. Izumi, T., Izumi, T., Kamei, S., Ooshita, F.: Randomized Gathering of Mobile Robots with Local-Multiplicity Detection. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 384–398. Springer, Heidelberg (2009)
16. Izumi, T., Izumi, T., Kamei, S., Ooshita, F.: Mobile Robots Gathering Algorithm with Local Weak Multiplicity in Rings. In: Patt-Shamir, B., Ekim, T. (eds.) SIROCCO 2010. LNCS, vol. 6058, pp. 101–113. Springer, Heidelberg (2010)
17. Kamei, S., Lamani, A., Ooshita, F., Tixeuil, S.: Asynchronous Mobile Robot Gathering from Symmetric Configurations without Global Multiplicity Detection. In: Kosowski, A., Yamashita, M. (eds.) SIROCCO 2011. LNCS, vol. 6796, pp. 150–161. Springer, Heidelberg (2011)
18. Kamei, S., Lamani, A., Ooshita, F., Tixeuil, S.: Asynchronous mobile robot gathering from symmetric configurations without global multiplicity detection. In: Proceedings of the 37th International Symposium on Mathematical Foundations of Computer Science (MFCS). Springer (to appear, 2012)
19. Klasing, R., Kosowski, A., Navarra, A.: Taking advantage of symmetries: Gathering of many asynchronous oblivious robots on a ring. Theor. Comput. Sci. 411, 3235–3246 (2010)
20. Klasing, R., Markou, E., Pelc, A.: Gathering asynchronous oblivious mobile robots in a ring. Theor. Comput. Sci. 390, 27–39 (2008)
21. Koren, M.: Gathering small number of mobile asynchronous robots on ring. Zeszyty Naukowe Wydzialu ETI Politechniki Gdanskiej. Technologie Informacyjne 18, 325–331 (2010)
22. Suzuki, I., Yamashita, M.: Distributed anonymous mobile robots: Formation of geometric patterns. SIAM J. Comput. 28(4), 1347–1363 (1999)
23. Yamashita, M., Souissi, S., Défago, X.: Gathering two stateless mobile robots using very inaccurate compasses in finite time. In: Proc. of the 1st int. Conf. on Robot Communication and Coordination (RoboComm), pp. 48:1–48:4 (2007)

Position Discovery for a System of Bouncing Robots

Jurek Czyzowicz[1], Leszek Gąsieniec[2], Adrian Kosowski[3], Evangelos Kranakis[4], Oscar Morales Ponce[4], and Eduardo Pacheco[4]

[1] Université du Québec en Outaouais, Gatineau, Québec J8X 3X7, Canada
[2] University of Liverpool, Liverpool L69 3BX, UK
[3] INRIA Bordeaux Sud-Ouest, LaBRI, 33400 Talence, France
[4] Carleton University, Ottawa, Ontario K1S 5B6, Canada

Abstract. A collection of n anonymous mobile robots is deployed on a unit-perimeter ring or a unit-length line segment. Every robot starts moving at constant speed, and bounces each time it meets any other robot or segment endpoint, changing its walk direction. We study the problem of *position discovery*, in which the task of each robot is to detect the presence and the initial positions of all other robots. The robots cannot communicate or perceive information about the environment in any way other than by bouncing. Each robot has a clock allowing it to observe the times of its bounces. The robots have no control on their walks, which are determined by their initial positions and the starting directions. Each robot executes the same *position detection algorithm*, which receives input data in real-time about the times of the bounces, and terminates when the robot is assured about the existence and the positions of all the robots.

Some initial configuration of robots are shown to be *infeasible* — no position detection algorithm exists for them. We give complete characterizations of all infeasible initial configurations for both the ring and the segment, and we design optimal position detection algorithms for all feasible configurations. For the case of the ring, we show that all robot configurations in which not all the robots have the same initial direction are feasible. We give a position detection algorithm working for all feasible configurations. The cost of our algorithm depends on the number of robots starting their movement in each direction. If the less frequently used initial direction is given to $k \leq n/2$ robots, the time until completion of the algorithm by the last robot is $\frac{1}{2}\lceil \frac{n}{k} \rceil$. We prove that this time is optimal. By contrast to the case of the ring, for the unit segment we show that the family of infeasible configurations is exactly the set of so-called *symmetric configurations.* We give a position detection algorithm which works for all feasible configurations on the segment in time 2, and this algorithm is also proven to be optimal.

1 Introduction

A mobile robot is an autonomous entity with the capabilities of *sensing*, i.e. ability to perceive some parameters of the environment, *communication* - ability to receive/transmit information to other robots, *mobility* - ability to move within the environment, and *computation* - ability to process the obtained data. Mobile robots usually act in a distributed way, i.e. a collection of mobile robots is deployed across the territory and they collaborate in order to achieve a common goal by moving, collecting and exchanging the data of the environment. The typical applications are mobile software agents (e.g. moving around and updating information about a dynamically changing network) or physical mobile robots (devices, robots or nano-robots, humans).

In many distributed applications, mobile robots operate in large collections of massively produced, cheap, tiny, primitive entities with very restricted communication, sensing and computational capabilities, mainly due to the limited production cost, size and battery power. Such groups of mobile robots, called *swarms*, often perform exploration or monitoring tasks in hazardous or hard to access environments. The usual swarm robot attributes assumed for distributed models include anonymity, negligible dimensions, no explicit communication, no common coordinate system (cf. [14]). Moreover, some of these models may assume obliviousness, limited visibility of the surrounding environment and asynchronous operation. In most situations involving such weak robots the fundamental research question concerns the feasibility of solving the given task (cf. [7, 11, 14]). When the question of efficiency is addressed, the cost of the algorithm is most often measured in terms of length of the robot's walk or the time needed to complete the task. This is also the case of the present paper, despite the fact that the robot does not have any control on its walk. In our case, the goal is to stop the robot's walk, imposed by the adversary, at the earliest opportunity - when the collected information (or its absence) is sufficient to produce the required solution.

Although the most frequently studied question for mobile robots is environment exploration, numerous papers related to such weak robots often study more basic tasks, such as pattern formation ([11, 13–15]). Gathering or point convergence ([5, 10]) and spreading (e.g. see [4]) also fall into this category. [14] introduced anonymous, oblivious, asynchronous, mobile robots which act in a so-called *look-compute-move* cycle. An important robot sensing capacity associated with this model permits to perceive the entire ([11, 13, 14]) or partial ([1, 10]) environment.

Contrary to the above model, in our paper, the robot has absolutely no control on its movement, which is determined by the bumps against its neighbors or the boundary points of the environment. In [2, 3] the authors introduced *population protocols*, modeling wireless sensor networks by extremely limited finite-state computational devices. The agents of population protocols also move according to some mobility pattern totally out of their control and they interact randomly in pairs. This is called *passive mobility*, intended to model, e.g., some unstable environment, like a flow of water, chemical solution, human blood, wind or unpredictable mobility of agents' carriers (e.g. vehicles or flocks of birds). In the recent work [12], a coordination mechanism based on meetings with neighboring robots on the ring was considered, also aiming at location discovery. The approach of [12] is randomized and the robots operate in the discrete environment in synchronous rounds.

Pattern formation is sometimes considered as one of the steps of more complex distributed task. Our involvement in the problem of this paper was motivated by the patrolling problem [6], where spreading the robots evenly around the environment may result in minimizing the *idleness* of patrolling, i.e., the time interval during which environment points remain unvisited by any robot. Clearly, position discovery discussed in the present paper is helpful in uniform spreading of the collection. A related problem was studied in [4], where the convergence rate of uniform spreading in one-dimensional environment in synchronous and semi-synchronous settings was discussed. Previously, [8] studied the problem of n robots $\{0, 1 \ldots, n-1\}$, initially placed in arbitrary order on the ring. It was shown that the rule of each robot i moving to the middle point between $i-1$ and $i+1$ may fail to converge to equal spreading (it was also shown in [8] that the system would converge if a fair scheduler activates units sequentially).

The model adopted in our paper assumes robot anonymity, passive mobility (similarly to that adopted in [2, 3]), restricted local sensing through bounce perception with a neighbor robot only, no communication between the robots, and continuous time. The only ability of the robot is the tacit observation of the timing of bounces and the computation and reporting of robots' locations. The private clock of each robot turns out to be a very powerful resource permitting to solve the problem efficiently in most cases.

2 The Model and Our Results

We consider a continuous, connected, one-dimensional universe in which the robots operate, which is represented either by a unit-perimeter ring or by a unit-length line segment. The ring is modeled by a real interval $[0, 1]$ with 0 and 1 corresponding to the same point. A set of n robots $r_0, r_1, \ldots, r_{n-1}$ is deployed in the environment and start moving at time $t = 0$ (where the indexing of the robots is used for purposes of analysis, only). The robots are not aware of the original positions and directions of other robots or the total number of robots in the collection. The robots move at constant unit speed, each robot starting its movement in one of the two directions. Each robot knows the perimeter of the ring (or the length of the segment) and it has a clock permitting to register the time of each of its bounces and store it in its memory. We assume that the time and distance travelled are commensurable, so during time t each robot travels distance t. Consequently, in the paper we compare distances travelled to time intervals.

By $r_i(t) \in [0, 1]$ we denote the position of robot r_i at time t. We suppose that originally each robot r_i occupies point $r_i(0)$ of the environment and that $0 \le r_0(0) < r_1(0) < \ldots < r_{n-1}(0) < 1$. Each robot is given an initial direction (clockwise or counterclockwise in the ring and left-to-right or right-to-left on the segment) at which it starts its movement. By dir_i we denote the starting direction of robot r_i and we set $dir_i = 1$ if r_i starts its movement in the counterclockwise direction around the ring or the left-to-right direction along the segment. By $dir_i = -1$ we denote the clockwise starting direction (on the ring) or right-to-left (on the segment). We call the sequence of pairs $(r_0(0), dir_0), \ldots, (r_{n-1}(0), dir_{n-1})$ the *initial configuration* of robots.

When two robots meet, they *bounce*, i.e., they reverse the directions of their movements. We call the trajectory of a robot a *bouncing walk*. The robots have no control on their bouncing walks, which depend only on their initial positions and directions, imposed to them by an adversary, and the bounces caused by meeting other robots. Each robot has to report the coordinates of all robots of the collection, i.e., their initial positions and their initial directions. The robots cannot communicate in any other way except for observing their meeting times. Each robot is aware of the type of the environment (ring or segment). All robots are anonymous, i.e. they have to execute the same algorithm. The only information available to each robot is the *bounce sequence*, i.e. the series of time moments $t_1, t_2, \ldots$, corresponding to its bounces resulting from the meetings with other robots.

By *position detection algorithm* we mean a procedure executed by each robot, during which the robot performs its bouncing walk and uses its bounce sequence as the data of the procedure, outputting the initial positions and directions of all robots. By the *cost* $C_{\mathcal{A}}(n)$ of algorithm $\mathcal{A}$ we understand the smallest value, such that for any feasible initial configuration of n robots in the environment, each robot executing $\mathcal{A}$ can report the initial configuration while performing a bouncing walk of total distance $C_{\mathcal{A}}(n)$. As in some cases the cost of the algorithm varies, depending on the robot initial directions, we denote by $C_{\mathcal{A}}(n, k)$ the cost of $\mathcal{A}$ for the class of initial configurations such that $1 \leq k \leq n/2$ robots start in one direction and $n - k$ start in the opposite one.

Question: Is it possible for each robot to find out, after some time of its movement, what is the number of robots in the collection and their relative positions in the environment? If not, what are the configurations of robots' initial positions and directions for which a position detection algorithm exists (i.e. it is possible to report the initial configuration after a finite time)? What is the smallest amount of time after which a robot is assured to identify all other robots in the collection?

Our goal is to propose an algorithm to be executed by any robot, which computes the original positions of all other robots of the collection. We say that such an algorithm is optimal if the time interval after which the robot is assured to have the knowledge of the positions of all other robots is the smallest possible.

We characterize all the feasible configurations for the ring and the segment. For both cases we give optimal position detection algorithms for all feasible configurations. Our algorithm for the segment requires $O(n)$ robot's memory, while constant size memory is sufficient for robots bouncing on the ring. [We suppose that in one memory word we may store a real value representing the robot's position in segment $[0, 1]$.]

For the case of the ring, we show that all robot configurations with not all robots given the same initial direction are feasible. We give a position detection algorithm working for all feasible configurations. The cost of our algorithm is not constant, but it depends on the number of robots starting their movement in each direction. When $k \leq n/2$ is the number of robots starting their walks in one direction with $n-k$ given the opposite direction we prove that our algorithm has cost $\frac{1}{2}\lceil\frac{n}{k}\rceil$. We prove that this algorithm is optimal.

For the case of the segment we prove that no position detection algorithm exists for *symmetric* initial configurations. Each symmetric configuration is a configuration of a subset of robots on a subsegment, concatenated alternately with its reflected copy and itself. We give a position detection algorithm of cost 2 working for all feasible (non-symmetric) configurations on the segment. This algorithm is proven to be optimal.

In Section 3 we give the position detection algorithm for the ring and prove its correctness for all feasible configurations. Section 4 analyses the cost of the position detection algorithm for the ring and proves its optimality. The segment environment is addressed in Section 5. The argument for the segment proceeds by reduction to that for the ring, but the criteria for a feasible configuration on the segment take a different form, dependent on the symmetry of the configuration.

3 The Algorithm on the Ring

As there is no system of coordinates on the ring common to all robots, each robot must compute the relative positions of other robots with respect to its own starting position. We may then infer that each robot assumes that its starting position is the point 0. We then suppose that $0 = r_0(0) < r_1(0) < \ldots < r_{n-1}(0) < 1$ and it is sufficient to produce the algorithm for robot r_0.

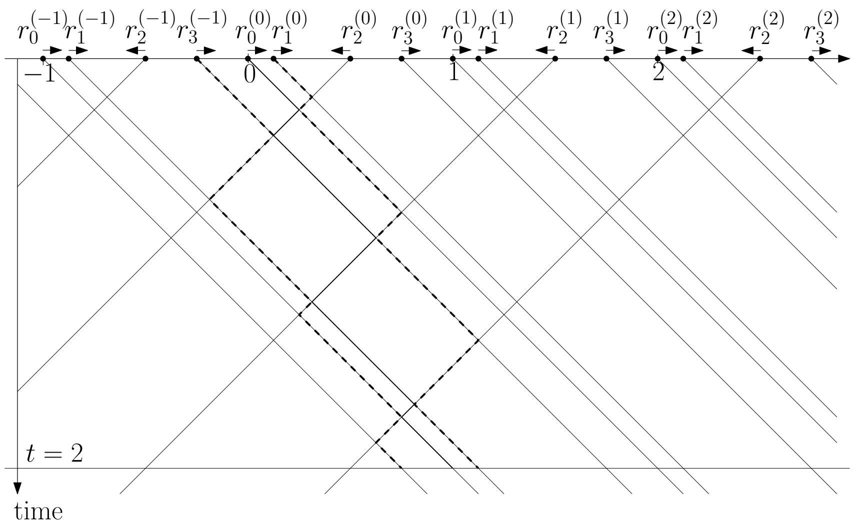
We assume in this paper that all robot indices are taken modulo n. When two robots meet, they reverse the directions of their movements, so the circular order of the robots around the ring never changes. When two robots r_i and r_{i+1} meet at time t, we have $r_i(t) = r_{i+1}(t)$, and $r_i(t)$ was moving counterclockwise while $r_{i+1}(t)$ was moving clockwise just before the meeting time t.

We denote by $dist(x, y)$ the distance that x has to traverse in the counterclockwise direction around the ring to reach the position of y (we call it the *counterclockwise distance* from x to y. Note that the *clockwise distance* from x to y equals $1 - dist(x, y)$.

In order to analyze the ring movement of the robots we consider an infinite line $L = (-\infty, \infty)$ and for each robot r_i, $0 \leq i \leq n-1$ we create an infinite number of its copies $r_i^{(j)}$, all having the same initial direction, such that their initial positions are $r_i^{(j)}(0) = j + r_i(0)$ for all integer values of $j \in \mathbb{Z}$ (see Fig. 1). We show that, when all copies of robots move along the infinite line while bouncing at the moments of meeting, all copies $r_i^{(j)}$ of a robot r_i bounce and reverse their movements at the same time. More precisely we prove

Lemma 1. *For all $t \geq 0, 0 \leq i \leq n-1$ and $j \in \mathbb{Z}$ we have $r_i^{(j+1)}(t) = r_i^{(j)}(t) + 1$.*

We use the concept of a *baton*, applied recently in [12]. Suppose that each robot initially has a virtual object (baton), that the robot carries during its movement,



$r_0^{(-1)}$ $r_1^{(-1)}$ $r_2^{(-1)}$ $r_3^{(-1)}$ $r_0^{(0)}$ $r_1^{(0)}$ $r_2^{(0)}$ $r_3^{(0)}$ $r_0^{(1)}$ $r_1^{(1)}$ $r_2^{(1)}$ $r_3^{(1)}$ $r_0^{(2)}$ $r_1^{(2)}$ $r_2^{(2)}$ $r_3^{(2)}$
−1
0
1
2
$t = 2$
time

Fig. 1. Example of a bouncing movement of four robots

but at the moment of meeting, two robots exchange their batons. By $b_i^{(j)}$ we denote the baton originally held by robot $r_i^{(j)}$ and by $b_i^{(j)}(t)$ we denote the position of this baton on the infinite line at time t. We can easily show the following lemma.

Lemma 2. *For all $t \geq 0, 0 \leq i \leq n-1$ and $j \in \mathbb{Z}$ we have $b_i^{(j)}(t) = b_i^{(j)}(0) + dir_i \cdot t = b_i^{(0)}(0) + j + dir_i \cdot t$.*

Proof. Since the bouncing robots exchange the batons, the batons travel at constant speed 1 in their original directions. Therefore, at time t each baton travelled the distance t so we have $b_i^{(j)}(t) = b_i^{(j)}(0) + dir_i \cdot t$. On the other hand, by construction we have $b_i^{(j+1)}(0) = b_i^{(j)}(0) + 1$ and both batons $b_i^{(j)}, b_i^{(j+1)}$ travel at unit speed in the same direction. Hence, we have by induction on j, that $b_i^{(0)}(t) = b_i^{(j)}(t) + j$. The claim of the lemma follows. ■

In Fig. 1 the trajectories of all the batons held originally by the robots going in direction dir are the lines of slope dir. Each robot r_i bounces while its trajectory intersects a trajectory of some baton, since this baton is then held by one of the robots r_{i-1}, r_{i+1}. For example, the trajectory of robot $r_0^{(0)}$, is represented by a fat polyline on Fig. 1, while the trajectories of its neighbor robots $r_3^{(-1)}$ and $r_1^{(0)}$ bouncing at $r_0^{(0)}$ are given by dashed polylines.

Lemma 3. *Consider robot r_a, which at the time moment t, while traveling in direction dir, meets some other robot. Suppose that, at the time of this meeting, r_a travelled the total distance d in direction dir (hence the total distance of $t-d$ in direction $-dir$). Then there exists a robot r_b, which was originally positioned at distance $(2d \mod 1)$ in direction dir on the ring. More precisely, $(2d \mod 1) =$*

$dist(r_a, r_b)$ if $dir = 1$ and $(2d \mod 1) = dist(r_b, r_a) = 1 - dist(r_a, r_b)$ if $dir = -1$. Moreover r_b started its movement in direction $-dir$.

Proof idea: Robot r_a and the baton of robot r_b approach (at speed 2) only when going in opposite directions, hence the time of this approach equals half of their initial distance.

Remark 1. The value $(2d \mod 1)$ may sometimes be equal to zero which corresponds to r_a meeting the robot currently holding the original baton of r_a (e.g. the sixth bounce of $r_0^{(0)}$ on Fig. 1). On the other hand, some meetings of robots may correspond to the same computed value of $(2d \mod 1)$ (e.g. all odd-numbered bounces of $r_0^{(0)}$ on Fig. 1), so some bounces do not have a new informative value about other robot positions.

The algorithm `RingBounce` executed by a robot, which reports initial positions and directions of all other robots on the ring, uses Lemma 3. Each bounce results in the output of information concerning one robot of the ring. In this way, a robot running such an algorithm needs only a constant-size memory. An additional test is made in line 10 to avoid outputting the same robot position more than once.

The robot's memory consists of two real variables *right* and *left* in which the robot will store the total distance travelled, respectively, in the counterclockwise and clockwise direction. The robot also accesses its system variable *clock* which automatically increases proportionally to the time spent while traveling (i.e. to the distance travelled).

```
Algorithm RingBounce ($dir$ : $\{-1, 1\}$);
1.   var $left \leftarrow 0, right \leftarrow 0$ : real;
2.   reset $clock$ to 0;
3.   while $true$ do
4.       do walk in direction $dir$ until
5.           (($clock - left \geq 1/2$) and ($clock - right \geq 1/2$)) or a meeting occurs;
6.       if ($clock - left \geq 1/2$) and ($clock - right \geq 1/2$) then EXIT;
7.       if $dir = 1$ then
8.           $right \leftarrow clock - left$;
9.           if ($0 < right < 1/2$) then
10.              OUTPUT robot at original position $2 \cdot right$ and direction $-dir$;
11.      else
12.          $left \leftarrow clock - right$;
13.          if ($0 < left < 1/2$) then
14.              OUTPUT robot at original position $1 - 2 \cdot left$ and direction $dir$;
15.      $dir \leftarrow -dir$;
```

Theorem 1. *Suppose that among all robots bouncing on the ring there is at least one robot having initial clockwise direction and at least one robot with the initial counterclockwise direction. The algorithm* `RingBounce`*, executed by any* *robot of the collection, correctly reports the initial positions and directions of all robots on the ring with respect to its initial position.*

Proof. Suppose w.l.o.g., that the robot executing `RingBounce` is robot r_0. Since there exists at least one other robot starting in the direction different from dir_0, robot r_0 will alternately travel in both directions, indefinitely bouncing against its neighbors r_1 and r_{n-1} on the ring. We show by induction, that at the start of each iteration of the while loop from line 3, the variable $left$ (resp. $right$) equals to the total distance travelled by r_0 clockwise (resp. counterclockwise). Suppose, by symmetry, that r_0 walks counterclockwise in the i-th iteration and the inductive hypothesis is true at the start of this iteration. Since, by inductive hypothesis, variable $left$ keeps the correct value through i-th iteration, variable $right$ is correctly modified at line 8, as $clock$ value equals the total distance travelled in both directions. Consequently, the inductive claim is true in the $(i + 1)$-th iteration.

We prove now that positions and directions of all robots are correctly reported before the algorithm ends. Take any robot r_i, $1 \leq i \leq n-1$. We consider first the case when the initial direction of r_i was clockwise. The trajectory of its original baton $b_i^{(0)}$ is then a line of slope 1 (cf. Fig. 1). Observe that robot r_0 stays at the same distance from baton b_i when walking in the clockwise direction and approaches it (reducing their counterclockwise distance $dist(r_0, b_i)$) when walking counterclockwise. Since $dist(r_0, b_i) \leq 1$, and r_0 and b_i walk towards each other, they approach at speed 2 during the counterclockwise movement of r_0. Consequently, the trajectories of r_0 and b_i intersect and r_0 eventually meets robot r_1 carrying baton b_i. Indeed, in line 4 of algorithm `RingBounce`, robot r_0 continues its movement as long as its total distance travelled in the counterclockwise direction is less than 1/2, which leads to the meeting of r_0 and r_1 (carrying baton b_i), before both robots finish their executions of the algorithm. Consequently, at the moment of their meeting, r_0 outputs at line 10 the initial distance between $r_0^{(0)}$ and $r_i^{(0)}$ on line L, which equals twice the time spent while the robots were approaching each other. As r_0 may obtain a copy of the same baton more than once (cf. r_0 intersecting several trajectories of batons $b_2^{(j)}$ on Fig. 1), the condition $(0 < right < 1/2)$ at line 9 permits to report the position of each other robot once only. Indeed, only $r_i^{(0)}$ - the copy of r_i at the closest counterclockwise distance to r_0 verifies this condition.

Consider now the case when robot r_i, $1 \leq i \leq n - 1$, starts its walk on the ring in the counterclockwise direction. Then r_0 obtains baton b_i while walking clockwise, i.e. at the moment of some bounce at r_{n-1}, while r_{n-1} holds baton b_i. In this case, robot r_0 stays at the same distance from baton b_i when walking in counterclockwise direction and approaches it (reducing their distance of $dist(b_i, r_0) = 1 - dist(r_0, b_i)$) when walking clockwise. At the moment when r_0 meets r_{n-1} holding baton b_i (whose trajectory originates from segment $[-1, 0]$ of L) the value of variable $left$ equals half the clockwise distance from $r_0(0)$ to $r_i(0)$. Indeed, at the moment of the meeting, half of this distance was covered by r_0 walking clockwise (the value of $left$) and the other half was covered by the counterclockwise move of baton b_i. Consequently the clockwise distance from the initial position of r_0 to the initial position of r_i equals $1 - 2 \cdot left$, correctly output at line 14. ■

Observe that, once the original positions and directions of all robots are reported, it is easy to monitor all further movements of all robots of the collection, i.e. their relative positions at any moment of time. However, this would require a linear memory of the robot performing such task.

4 The Execution Time of Bouncing on the Ring

As stated in the introduction, we look for the algorithm of the optimal cost, i.e. the smallest possible total distance travelled, needed to correctly report any initial configuration. We show that the algorithm `RingBounce` is the optimal one, i.e. that the time moment, at which the robot can be sure that the positions of all other robots have been reported, is the time when the robot stops executing `RingBounce`. Observe that algorithm `RingBounce` has cost at least 1, i.e. a robot executing it must travel at least distance 1. Indeed, the loop from lines 4-5 continues unless robot's walk distance in each direction totals at least half the size of the ring. On the other hand, the example from Fig. 1 shows, that if the number of robots starting their walks in one direction is different from the number of robots starting walking in the opposite direction, the total cost of `RingBounce` may be higher. We have

Theorem 2. *Consider a collection of n robots on the ring, such that k of them, $1 \leq k \leq n/2$, have one initial direction and the remaining $n-k$ robots have the other initial direction. Then the cost of* `RingBounce` *is $C_{\mathcal{RB}}(n,k) \leq \frac{1}{2}\lceil\frac{n}{k}\rceil$.*

Proof idea: If there are more robots starting in one direction, say positive direction dir, than in direction $-dir$ then r_i gets more frequently dir-moving batons (cf. Fig. 1). Since the route of r_i, intersects the trajectory of each baton only once, r_i must meet copies of batons originating from other segments than $[0,1]$ of line L. By counting we show that the last such segment is $[\lceil(n-k)/k\rceil - 1, \lceil(n-k)/k\rceil]$. Hence, in the worst case, r_i walks distance $1/2$ in direction dir and distance $\lceil(n-k)/2k\rceil$ in direction $-dir$.

From Theorem 2 we immediately have the following Corollary, which bounds the worst-case walking time for a robot.

Corollary 1. *Assuming that the collection of n robots admits robots starting their movements in both directions around the ring, Then the cost of* `RingBounce` *is $C_{\mathcal{RB}}(n) \leq \frac{n-1}{2}$.*

The algorithm `RingBounce` continues until the total lengths of walks in both directions reach the values of at least $1/2$, since this guarantees that the presence of each robot is eventually detected. The following theorem proves that the cost of `RingBounce` algorithm is optimal even if the (a priori) knowledge of the number of robots is assumed.

Theorem 3. *Suppose that there is a collection of n robots on the ring, such that k of them, $1 \leq k < n/2$, have one initial direction and the remaining $n-k$ robots have the other initial direction. Then for every $\epsilon > 0$ there exists a distribution of such robots on the ring with their initial positions $0 \leq r_0 < r_1 < \ldots < r_{n-1} < 1$, so that a position detection algorithm terminating at time $\frac{1}{2}\lceil \frac{n}{k} \rceil - \epsilon$ cannot determine the initial positions of all robots on the ring, even if the values of n and k are known in advance.*

Proof idea: Following the argument from the proof of Theorem 2 we put in the ring the last of the $n-k$ agents starting in direction dir such that r_i is forced to get a baton originating arbitrarily close to point $\lceil (n-k)/k \rceil$ of line L. This requires r_i to walk the total distance arbitrarily close to $\frac{1}{2}\left\lceil \frac{k}{n} \right\rceil$.

Clearly each configuration of robots with the same initial direction of all robots is infeasible, because no robot ever bounces. Consequently from Theorem 2 and Theorem 3 follows

Corollary 2. *The family of infeasible initial configurations of robots on the ring contains all configurations with the same initial direction of all robots.* `RingBounce` *is the optimal position detection algorithm for all feasible initial configurations of robots on the ring. This algorithm assumes constant-size memory of the robot running it.*

Clearly, we can easily adapt algorithm `RingBounce`, so for infeasible initial configuration the algorithm stops and reports the infeasibility. It is sufficient to test whether the very first walk of the robot ends with a bounce before the robot traverses the distance of $1/2$.

5 Bouncing on the Line Segment

In this section we show how the algorithm for bouncing robots may be used for the case of a segment. We suppose that each robot walks along the unit segment changing its direction when bouncing from another robot or from an endpoint of the segment. Robots have the same capabilities as in the case of the ring and they cannot distinguish between bouncing from another robot and bouncing from a segment endpoint.

We consider the segment $[0, 1)$ containing n robots, initially deployed at positions $0 \leq r_0(0) < r_1(0), \ldots, r_{n-1}(0) < 1$. Each robot r_i, $0 \leq i \leq n-1$ is given an initial direction dir_i, such that $dir_i = 1$ denotes the left to right initial movement and $dir_i = -1$ denotes initial movement from right to left on segment $[0, 1)$. The robots start moving with unit speed at the same time moment $t = 0$ at the predefined directions and they change direction upon meeting another robot or bumping at the segment endpoint. The main difficulty of the segment case is that the robot r executing the position detection algorithm for the ring has to report the *relative* locations of other robots, i.e. their distances to its own initial position $r(0)$, while in the segment case the *absolute* distance from $r(0)$ to the segment endpoint has to be found.

We show in this section that the bouncing problem is feasible for all initial robot configurations except a small set of symmetric ones. Intuitively, an initial configuration of robots is *symmetric* if the unit segment may be partitioned into k subsegments $S_0, S_1, \dots, S_{k-1}$, such that the positions and directions of robots in each subsegment form a reflected copy of positions and directions of robots in a neighboring subsegment (see Fig. 2). More formally we have the following

Definition 1. *A configuration* $C = ((r_0(0), dir_0), \dots, (r_{n-1}(0), dir_{n-1}))$ *is symmetric if there exists a positive integer* $k < n$*, such that* $n \mod k = 0$ *and the partition of segment* $S = [0, 1)$ *into subsegments* $S_0 = [0, \frac{1}{k})$, $S_1 = [\frac{1}{k}, \frac{2}{k}), \dots, S_1 = [\frac{k-1}{k}, 1)$ *with the following property. For each robot* r_i, $0 \le i < n$*, if* $r_i(0) = \frac{p}{n} + x$*, for* $0 \le x < \frac{1}{k}$*, (i.e.* $r_i(0) \in S_p$*),* $0 \le p < n$*, then, if* $p > 0$*, there exists a robot* $r_{i'}$*, such that* $r_{i'}(0) = \frac{p}{n} - x$ *and* $dir_{i'} = 1 - dir_i$ *and, if* $p < n - 1$*, there exists a robot* $r_{i''}$*, such that* $r_{i''}(0) = \frac{p+2}{n} - x$ *and* $dir_{i''} = 1 - dir_i$*.*

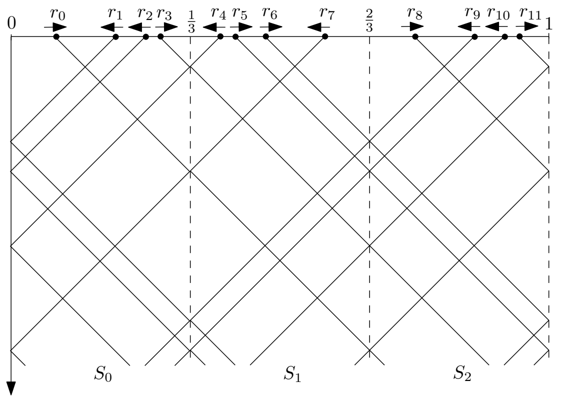


Fig. 2. Example of a symmetric initial configuration of $n = 12$ robots containing $k = 3$ subsegments

Theorem 4. *Every symmetric initial configuration of robots is infeasible.*

Proof. Let $C_1 = ((r_0(0), dir_0), \dots, (r_{n-1}(0), dir_{n-1}))$ be a symmetric configuration and k the number of consecutive subsegments, each one being the reflected copy of its neighbor. Construct now configuration $C_2 = ((r'_0(0), dir'_0), \dots, (r'_{n-1}(0), dir'_{n-1}))$ of n robots considering also a sequence of n equal size intervals and swapping the roles of odd-numbered and even-numbered robots of C_1. More precisely for each robot r_i, such that $r_i(0) \in S_p = [\frac{p}{n}, \frac{p+1}{n})$ there exists a robot r'_j such that $r'_j(0) = \frac{2p+1}{n} - r_i(0)$ and $dir'_j = 1 - dir_i$. Observe that, no robot ever crosses the boundary of any subsegment S_p, i.e. $r_i(0) \in S_p$ implies $r_i(t) \in S_p$, for any $t \ge 0$. Indeed, by construction, for any robot reaching and endpoint of S_p, different from points 0 and 1, at the same time moment there is another robot approaching this endpoint from the other side within the reflected copy of S_p provoking a bounce (cf. Fig. 3). Therefore, within each even-numbered subsegment S_{2n} of a symmetric configuration the relative positions of robots and their directions are the same (similarly within each odd-numbered subsegment). Consequently, no robot can distinguish whether it is, say, in an even-numbered segment of C_1 or in an odd-numbered segment of C_2 so its position in segment $[0, 1)$ is unknown. ■

We show now how the position detection algorithm for the ring may be used in the case of the segment.

Let S be a unit segment containing n robots at initial positions $r_0(0) < r_1(0) < \ldots < r_{n-1}(0)$ and the initial directions $dir_0, \ldots, dir_{n-1}$. Suppose that a segment $S^R \subset [1, 2]$ is the reflected copy of S containing n robots $r_n^R, \ldots, r_{2n-1}^R$ at the initial positions $r_n^R(0) = 2 - r_n(0) < r_{n-1}^R(0) = 2 - r_{n-1}(0) < \ldots < r_0^R(0) = 2 - r_0(0)$. The initial directions of each robot r_i^R is $1 - dir_i$ for $0 \le i < n$. Let R_2 be the ring of perimeter 2 composed of segment S concatenated with segment S^R, with points 0 and 2 identified.

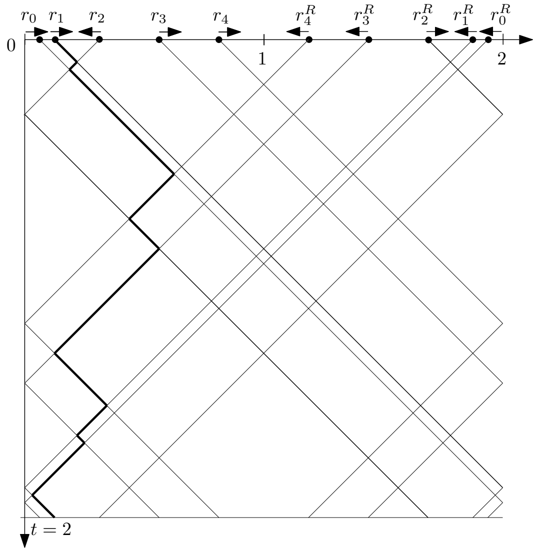


Fig. 3. Five robots on a segment $[0, 1)$ and their reflected copy

Consider the walk of robots r_i, for $0 \le i < n$, within segment S and ring R_2. Let $t_0 = 0$ and $0 \le t_1 < t_2 \ldots$ be the sequence of time moments during which some bounces occur. Each such bounce takes place either between some pair of robots or when some robot bounces from an endpoint of S. It is easy to see by induction on i that at any time moment $t \in [t_i, t_{i+1}]$ each robot r_j has the same positions in S and R_2 as well as the same direction of movement and that the S^R part of R_2 is a reflected copy of S. Indeed, by construction, this condition is true for the interval $[t_0, t_1]$. If robots r_j, r_{j+1} bounce against each other in S at time t_i, at the same time robots r_j, r_{j+1} bounce in R_2, as well as, by symmetry r_j^R

bounces against r^R_{j+1}. If in time t_i robot r_0 (or r_{n-1}) bounce from an endpoint of S, by inductive hypothesis r_0 bounces against r^R_0 at point $0 \in R_2$ (or r_{n-1} bounces against r^R_{n-1} at point $1 \in R_2$). In each case, the inductive condition holds. We just showed

Lemma 4. *The bounce sequence of any robot r_i on segment S is the same as the bounce sequence of r_i on ring R_2.*

To prove that only symmetric configuration of robots on the segment are infeasible we need the following lemma.

Lemma 5. *Suppose that the initial configuration of robots $C = ((r_0(0), dir_0), \ldots, (r_{n-1}(0), dir_{n-1}))$ on a unit segment is not symmetric. Then no internal robot r_i, for $1 \le i \le n-2$, may have all its left bounces or all right bounces at the same point of the unit segment.*

Proof. Suppose, by contradiction that there exists an internal robot having all its left bounces at the same point (the proof in the case of all right bounces falling at the same point is similar, by symmetry). Let i be the smallest index $1 \le i \le n-2$ of a robot with this property and point x, $0 < x < 1$, be the point of all left bounces of r_i. We show first that the initial configuration of robots belonging to segment $[0, x]$ is the reflected copy of the initial configuration of robots belonging to segment $[x, 2x]$ Then robot r_{i-1} has all its right bounces also at point x. Consequently, at each moment of time after the first such bounce, the position and the direction of robot r_{i-1} is a symmetric (reflected) copy of robot r_i with respect to point x. Then, if $i \ge 2$, the trajectory of r_{i-2} is a reflected copy of the trajectory of r_{i+1}. By induction on i, for any $q \ge 0$ the trajectory of r_q is the reflected copy of the trajectory of r_{2i-q-1} and finally the trajectory of r_0 is the reflected copy of the trajectory of r_{2i-1}. Therefore, all right bounces of robot r_{2i-1} are at point $2x$ of the unit segment, so initial configuration of robots belonging to segment $[0, x]$ is the reflected copy of the initial configuration of robots belonging to segment $[x, 2x]$, as needed.

By induction on j we prove that each subsegment $[(j-1)x, jx]$ is a reflected copy of subsegment $[jx, (j+1)x]$. By minimality of x, no such subsegment contains a point which is never crossed by any robot, hence, for some value of j, we have $jx = 1$, concluding the proof. ■

We can show now, that the set of configurations on the unit segment for which no position detection algorithm exists is exactly the set of symmetric configurations. For all other configurations we propose an optimal position detection algorithm. We suppose that the robot assumes that its initial direction on the segment is positive. Otherwise, the robot needs to be *chirality aware*, i.e. capable of identifying the positive direction of the segment.

Theorem 5. *For any collection of n robots not in a symmetric initial configuration on the unit segment there exist a position detection algorithm $\mathcal{A}$ with cost $C_{\mathcal{A}}(n) = 2$. For any $\epsilon > 0$ there exist collections of robots, such that some of them cannot terminate the execution of any position detection algorithm before time $2 - \epsilon$.*

Proof idea: To construct $\mathcal{A}$ we first adapt algorithm `RingBounce` by scaling up by the factor of 2 all distance and time constants in order to make it work for R_2 - a ring of size 2. By Lemma 4 its output produces robots' configuration on a unit segment S and its reflected copy S^R. By non-symmetricity of configuration on $[0, 1]$ the subdivision of R_2 into S and S^R is unique and the robot positions within S are obtained (this needs 0(n) memory). Using Theorem 2 the total cost is 2, since n robots walked in each direction and the ring was of size 2. To prove cost optimality we take the two robots sufficiently close to point 0, walking in positive direction and observe that the first of them must walk the distance arbitrarily close to 1 in each direction, in order to find out that its first bounce was against the second robot rather that against the right endpoint, in order to identify the initial configuration.

As the algorithm for the segment, presented in the proof of Theorem 5 assumes storing in robot's memory the positions of all robots, from Theorems 4 and 5 follows

Corollary 3. *The family of infeasible initial configurations of robots on the segment contains all symmetric initial configurations of robots. There exists an optimal position detection algorithm for all feasible initial configurations of robots on the segment. This algorithm assumes $O(n)$-size memory of the robot executing it.*

6 Conclusion

The algorithms of the paper may be extended to the case when only one robot r_0 starts moving initially (while all other robot movements are triggered by bounces) and r_0 must report other robots' initial positions. Indeed, observe that all robots must be moving at no later than time 1 for the ring and at no later than time 2 for the segment. Robot r_0 may then compute the trajectories of all other robots as if they started moving simultaneously and then successively compute the sequence of motion triggering bounces of all robots.

One open problem is to determine whether there exists an optimal position detection algorithm for the segment using a constant size memory. Another open problem is whether the bouncing problem may be solved for the case of robots having different initial speeds. If we assume the momentum conservation principle, so that the bouncing robots exchange their speeds, the baton trajectories still remain semi-lines of constant slopes. Therefore, if each robot is always aware of its current speed, perhaps it might be possible, that, after a finite time, it learns the starting positions and initial speeds of all other robots.

The location discovery performed by the collection of robots, presented in this paper, may be used for the equally-spaced self-deployment of the robots around the environment (e.g. to perform optimal patrolling) or for some other pattern formation task. However, such a task would require an additional robot capacity besides passive mobility the way it is assumed in this paper. Once the positions of the entire collection is known, the robots need to synchronize their movements, e.g. by adding waiting periods.

Acknowledgements. Research of J. Czyzowicz and E. Kranakis supported in part by NSERC grants, L. Gąsieniec was sponsored by the Royal Society Grant IJP - 2010/R2 , O. Morales by MITACS grant and E. Pacheco by CONACyT and NSERC grant.

References

1. Ando, H., Oasa, Y., Suzuki, I., Yamashita, M.: Distributed memoryless point convergence algorithm for mobile robots with limited visibility. IEEE Transactions on Robotics and Automation 15(5), 818–828 (1999)
2. Angluin, D., Aspnes, J., Diamadi, Z., Fischer, M.J., Peralta, R.: Computation in networks of passively mobile finite-state sensors. In: Distributed Computing, pp. 235–253 (2006)
3. Angluin, D., Aspnes, J., Eisenstat, D.: Stably computable predicates are semilinear. In: Proc. of PODC, pp. 292–299 (2006)
4. Cohen, R., Peleg, D.: Local spreading algorithms for autonomous robot systems. Theoretical Computer Science 399(1-2), 71–82 (2008)
5. Cohen, R., Peleg, D.: Convergence Properties of the Gravitational Algorithm in Asynchronous Robot Systems. SIAM Journal on Computing 34(6), 1516–1528 (2005)
6. Czyzowicz, J., Gąsieniec, L., Kosowski, A., Kranakis, E.: Boundary Patrolling by Mobile Agents with Distinct Maximal Speeds. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 701–712. Springer, Heidelberg (2011)
7. Das, S., Flocchini, P., Santoro, N., Yamashita, M.: On the Computational Power of Oblivious Robots: Forming a Series of Geometric Patterns. In: Proc. of PODC, pp. 267–276 (2010)
8. Dijkstra, E.W.: Selected Writings on Computing: Personal Perspective, pp. 34–35. Springer, New York (1982)
9. Efrima, A., Peleg, D.: Distributed algorithms for partitioning a swarm of autonomous mobile robots. Theoretical Computer Science 410, 1355–1368 (2009)
10. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Gathering of asynchronous oblivious robots with limited visibility. Theor. Comput. Sci. 337(1-3), 147–168 (2005)
11. Flocchini, P., Prencipe, G., Santoro, N., Widmayer, P.: Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. Theor. Comput. Sci. 407(1-3), 412–447 (2008)
12. Friedetzky, T., Gąsieniec, L., Gorry, T., Martin, R.: Observe and Remain Silent (Communication-Less Agent Location Discovery). In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 407–418. Springer, Heidelberg (2012)
13. Sugihara, K., Suzuki, I.: Distributed algorithms for formation of geometric patterns with many mobile robots. Journal of Robotic Systems 13(3), 127–139 (1996)
14. Suzuki, I., Yamashita, M.: Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. SIAM J. Comput. 28(4), 1347–1363 (1999)
15. Yamashita, M., Suzuki, I.: Characterizing geometric patterns formable by oblivious anonymous mobile robots. Th. Comp. Science 411(26-28), 2433–2453 (2010)

Counting-Based Impossibility Proofs for Renaming and Set Agreement*

Hagit Attiya and Ami Paz

Department of Computer Science, Technion

Abstract. Renaming and set agreement are two fundamental sub-consensus tasks. In the *M-renaming* task, processes start with names from a large domain and must decide on distinct names in a range of size M; in the *k-set agreement* task, processes must decide on at most k of their input values. Renaming and set agreement are representatives of the classes of *colored* and *colorless* tasks, respectively.

This paper presents simple proofs for key impossibility results for wait-free computation using only read and write operations: n processes cannot solve $(n-1)$-set agreement, and, if n is a prime power, n processes cannot solve $(2n-2)$-renaming.

Our proofs consider a restricted set of executions, and combine simple operational properties of these executions with elementary counting arguments, to show the existence of an execution violating the task's requirements. This makes the proofs easier to understand, verify, and hopefully, extend.

1 Introduction

In a basic shared-memory system, n asynchronous processes communicate with each other by writing and reading from the memory. Solving a distributed *task* requires processes, each starting with an input, to decide on outputs that satisfy certain requirements. A *wait-free* algorithm for a task ensures that each process decides within a finite number of its own steps. In this model, we can only solve *sub-consensus* tasks, which are weaker than consensus but still provide some nontrivial coordination among processes. Two prime examples of sub-consensus tasks are renaming and set agreement.

In *M-renaming* [2], processes must choose distinct names in a range of size $M \geq n$. The range of new names, i.e., the value of M, determines the efficiency of algorithms that rely on the new names [4], and hence, it is important to minimize M. Clearly, n-renaming is trivial if processes may use their identifiers, such that process p_i chooses name i; to rule out such solutions, the algorithm is required to be *symmetric* [10, 29]. There are wait-free algorithms for $(2n-1)$-renaming [2, 4, 9, 21], and it has been argued that there is no wait-free algorithm for $(2n-2)$-renaming [5, 26–28].

* This research is supported in part by Yad-HaNadiv fund and the *Israel Science Foundation* (grant number 1227/10).

In *k-set agreement* [16], processes must decide on at most k of their input values. Clearly, n processes can solve n-set agreement, by each deciding on its input value; it has been proved that $(n-1)$-set agreement cannot be solved by a wait-free algorithm [8,27,32].

Our contribution: This paper presents simple proofs of the lower bounds for renaming and set agreement; the proofs consider a restricted set of executions and use counting arguments to show the existence of an execution violating the task's requirements. While there are several simple proofs for set agreement [1,3,5], such proofs for renaming have eluded researchers for many years.

Our main result is a proof for the impossibility of $(2n-2)$-renaming, when n is a power of a prime number. The proof goes by considering the *weak symmetry breaking* (*WSB*) task: in WSB, each process outputs a single bit, so that when all processes participate, not all of them output the same bit. A $(2n-2)$-renaming algorithm easily implies a solution to WSB, by deciding 1 if the new name decided by the process is strictly smaller than n, and 0 otherwise; therefore, the impossibility of $(2n-2)$-renaming follows from the impossibility of WSB. (There is also a reduction in the opposite direction [22], but it is not needed for the lower bound.)

We prove that WSB is unsolvable when n, the number of processes, is a power of a prime number, by showing that every WSB algorithm for this number of processes has an execution in which all processes output the same value. Since we are proving a lower bound, it suffices to consider a restricted set of executions (corresponding to *immediate atomic snapshot executions* [8,9] or *block executions* [5]). The existence of a "bad" execution, violating the task's requirements, is proved by assigning a *sign* to each execution, counting the number of bad executions by sign, and concluding that a "bad" execution exists.

Prior lower bound proofs for renaming [11–13] rely on concepts and results from combinatorial and algebraic topology. Although our proof draws on ideas from topology (see the discussion), it uses only elementary counting arguments and simple operational properties of block executions. This makes the proof easier to understand, verify, and hopefully, extend.

As a warm-up and to familiarize the reader with counting-based arguments, we prove the impossibility of wait-free solutions for $(n-1)$-set agreement, by counting the bad executions in two complementary ways. We believe this proof is interesting on its own due to its extreme simplicity.

Another intermediate result shows the impossibility of solving *strong symmetry breaking* (*SSB*)—an adaptive variant of WSB, in which at least one process outputs 1 in every execution. The impossibility proof for SSB is similar to the proof for WSB, although it is simpler—it holds for non-symmetric algorithms and for every value of n. SSB is closely related to an *adaptive* variant of renaming, in which M, the range of new names, depends only on the number of processes participating in the execution. Specifically, in $\left(2p - \lceil \frac{p}{n-1} \rceil\right)$-*adaptive renaming*, if $p < n$ processes participate in an execution, they choose distinct names in $1, \ldots, 2p-1$, and if all n processes participate they choose distinct names in $1, \ldots, 2n-2$. This task solves SSB, by deciding on 1 if the output of $\left(2p - \lceil \frac{p}{n-1} \rceil\right)$-adaptive renaming is strictly smaller than n and 0 otherwise.

Previous research: The impossibility of $(2n-2)$-renaming was proved by considering WSB [5,26–28]. All these papers claim that no algorithm solves WSB for any number of processes, using the same topological lemma. A few years ago, however, Castañeda and Rajsbaum [12,13] proved that this lemma is incorrect, and gave a different proof for the impossibility of WSB, which holds only if the binomial coefficients $\binom{n}{1}, \ldots, \binom{n}{n-1}$ are not relatively prime (see also [11]). It can be shown that these values of n are precisely the prime powers, indicating that the lower bound holds for a small fraction of the possible values of n.[1] For the other values of n, Castañeda and Rajsbaum gave a non-constructive proof, using a subdivision algorithm, for the existence of a WSB algorithm [12,15]. The upper and lower bound proofs use non-trivial topological tools on *oriented manifolds.*

The k-set agreement task is *colorless*, which intuitively means that the output of one process can be adopted by another process. Colorless tasks have been extensively studied: there is a complete characterization of their wait-free solvability by simple topological conditions [27,28], which implies that wait-free $(n-1)$-set agreement is impossible. Wait-free $(n-1)$-set agreement is also proved impossible by topological methods [5,8,26,32], or graph theory [1,3]. Set agreement and its variants have been studied in many other models as well, see, e.g., the survey in [31].

Clearly, renaming is not colorless, since the new name chosen by one process cannot be adopted by another one; such tasks are called *colored.* Little is known about colored tasks and only a few tasks were studied (e.g., [19,29]). A good survey of renaming can be found in [14]. We hope that our more direct treatment will encourage the investigation of colored tasks.

2 Preliminaries

We use a standard model of an asynchronous shared-memory system [6]. There is a set of n *processes* $\mathbb{P} = \{p_0, \ldots, p_{n-1}\}$, each of which is a (possibly infinite) state machine. Processes communicate with each other by applying read and write *operations* to shared registers. Each process p_i has an unbounded *single-writer multi-reader register* R_i it can write to, which can be read by all processes.

An *execution* of an algorithm is a finite sequence of read and write operations by the processes. Each process p_i starts the execution with an *input value*, denoted In_i, performs a computation and then *terminates* with an *output value*, also called a *decided value.* Without loss of generality, assume that in the algorithm, a process alternates between writing its complete state to its register and reading all the registers (performing a *scan*).

[1] Asymptotically, there are $\Theta\left(\frac{N}{\log N}\right)$ primes, and $\Theta\left(\sqrt{N} \log N\right)$ powers of primes with exponent $e \geq 2$ [23, pp. 27-28] in the interval $[1, N]$. Hence, the portion of prime powers is $\Theta\left(\frac{1}{\log N} + \frac{\log N}{\sqrt{N}}\right)$, which tends to 0 as N goes to ∞.

For a set of processes P, we say α is an execution *of* P if all processes in P take steps in α, and only them; P is the *participating set* of α. Although any process may fail during the execution, we restrict our attention to executions where every participating process terminates (possibly by taking steps after all other processes terminated).

For a process p_i that terminates in an execution α, the output of p_i in α is denoted $dec(\alpha, p_i)$. For a set of processes P, $dec(\alpha, P)$ is the set of outputs of the processes in P that terminate in α, and $dec(\alpha)$ is the set of all outputs in α.

Two executions α, α' are *indistinguishable* to process p_i, denoted $\alpha \overset{p_i}{\sim} \alpha'$, if the state of p_i after both executions is identical. We write $\alpha \overset{P}{\sim} \alpha'$, if $\alpha \overset{p_i}{\sim} \alpha'$ for every process $p_i \in P$.

A *block execution* [5], or *immediate atomic snapshot execution* [8,9], is induced by *blocks*, i.e., nonempty sets of processes. A block execution α is induced by a sequence of blocks $B_1 B_2 \cdots B_h$, if it begins with all processes in B_1 writing in an increasing order of identifiers and then performing a scan in the same order, followed by all processes of B_2 writing and then performing a scan, and so on. Since we prove impossibility results, we can restrict our attention to block executions. Note that given a block execution as a sequence of read and write operations, there is a unique sequence of blocks inducing the execution: each consecutive set of write operations determines a block.

We consider *wait-free* algorithms, in which each process terminates in a finite number of its own steps, regardless of the steps taken by other processes. Since only executions with a bounded set of inputs are considered, there is a common upper bound on the number of steps taken in all these executions.[2]

3 Impossibility of $(n-1)$-Set Agreement

The *k-set agreement* task is an extension of the *consensus* task, where processes have to decide on at most k values. Process p_i has an input value (not necessarily binary), and it has to produce an output value satisfying:

k-Agreement: At most k different values are decided.
Validity: Every decided value is an input value of a participating process.

A process p_i is *unseen* in an execution α if it takes steps in α only after all other processes terminate. In this case, α is induced by $B_1 \cdots B_h \{p_i\}\{p_i\}^*$, where $p_i \notin B_j, 1 \leq j \leq h$, and $\{p_i\}^*$ stands for a finite, nonnegative number of blocks of the form $\{p_i\}$.

A process p_i is *seen* in an execution α if it is not unseen in it. A process $p_i \in B_j$ is *seen* in B_j if p_i is not the only process in B_j, or if there is a later block with a process other than p_i. The key property of block executions that we use is captured by the next lemma (this is Lemma 3.4 in [5]).

[2] In general, wait-freedom does not necessarily imply that the executions are bounded [25]. However, with a bounded set of inputs, wait-freedom implies that there is a bound on the number of steps taken by a process in *any* execution.

Lemma 1. *Let P be a set of processes, and let $p_i \in P$. If p_i is seen in an execution α of P, then there is a unique execution α' of P such that $\alpha' \overset{P-p_i}{\sim} \alpha$ and $\alpha' \neq \alpha$. Moreover, p_i is seen in α'.*

Sketch of proof. Let α be induced by $B_1 \cdots B_h \{p_i\}^*$, and let B_ℓ be the last block in which p_i is seen.

If $B_\ell = \{p_i\}$, define the new execution α' by merging B_ℓ with the successive block $B_{\ell+1}$. That is, $\{p_i\}B_{\ell+1}$ is replaced with $\{p_i\} \cup B_{\ell+1}$ (note that $B_{\ell+1}$ does not include p_i), and all other blocks remain the same.

Otherwise, if $B_\ell \neq \{p_i\}$, define α' by splitting p_i before B_ℓ, with the opposite manipulation. That is, B_ℓ is replaced with $\{p_i\}B_\ell \setminus \{p_i\}$, and all other blocks remain the same.

In both cases, it might be necessary to add singleton steps at the end of the execution to ensure p_i terminates. □

Assume, by way of contradiction, that there is a wait-free algorithm solving $(n-1)$-set agreement. Let C_m, $1 \leq m \leq n$, be the set of all executions in which only the first m processes, $p_0, \ldots, p_{m-1}$, take steps, each process p_i has an input value i, and all the values $0, \ldots, m-1$ are decided. We prove that $C_n \neq \emptyset$, i.e., there is an execution in which n values are decided.

Lemma 2. *For every m, $1 \leq m \leq n$, the size of C_m is odd.*

Proof. The proof is by induction on m. For the base case, $m = 1$, consider a solo execution of p_0. Since the algorithm is wait-free, p_0 decides in h steps, for some integer $h \geq 1$. By the validity property, p_0 decides on 0, so there is a unique execution in C_1, induced by $\{p_0\}^h$. Hence, $|C_1| = 1$.

Assume the lemma holds for some m, $1 \leq m < n$. Let X_{m+1} be the set of all tuples of the form (α, p_i), $0 \leq i \leq m$, such that α is an execution where only processes $p_0, \ldots, p_m$ take steps, and all the values $0, \ldots, m-1$ are decided by processes other than p_i; p_i decides on an arbitrary value. We show that the sizes of X_{m+1} and C_{m+1} have the same parity.

Let X'_{m+1} be the subset of X_{m+1} containing all tuples (α, p_i), such that all values $0, \ldots, m$ are decided in α; we show that the size of X'_{m+1} is equal to the size of C_{m+1}. Let (α, p_i) be a tuple in X'_{m+1}, so α is in C_{m+1}. Since $m+1$ values are decided by $m+1$ processes in α, p_i is the unique process that decides m in α, so there is no other tuple (α, p_j) in X'_{m+1} with the same execution α. For the other direction, if α is an execution in C_{m+1}, then in α, $m+1$ values are decided by $m+1$ processes, and there is a unique process p_i which decides m in α. Hence, α appears in X'_{m+1} exactly once, in the tuple (α, p_i).

We next argue that there is an even number of tuples in X_{m+1}, but not in X'_{m+1}. If (α, p_i) is such a tuple, then p_i decides $v \neq m$ in α. Since $(\alpha, p_i) \in X_{m+1}$, all values but m are decided in α by processes other than p_i, so there is a unique process $p_j \neq p_i$ that decides v in α. Thus, (α, p_i) and (α, p_j) are both in X_{m+1} but not in X'_{m+1}, and these are the only appearances of α in X_{m+1}. Therefore, there is an even number of tuples in X_{m+1} but not in X'_{m+1}, implying that the sizes of X_{m+1} and X'_{m+1} have the same parity, and hence, the sizes of X_{m+1} and C_{m+1} have the same parity.

To complete the proof and show that the size of X_{m+1} is odd, partition the tuples (α, p_i) in X_{m+1} into three subsets, depending on whether p_i is seen in α or not:

1. p_i is **seen** in α: By Lemma 1, for each execution α in which only $p_0, \ldots, p_m$ take steps, and p_i is seen, there is a unique execution $\alpha' \neq \alpha$ with only $p_0, \ldots, p_m$ taking steps, p_i is seen, and all processes other than p_i decide on the same values. Hence, the tuples in X_{m+1} in which p_i is seen in the execution can be partitioned into disjoint pairs of the form $\{(\alpha, p_i), (\alpha', p_i)\}$, which implies that there is an even number of such tuples.
2. $\boldsymbol{i \neq m}$ and p_i is **unseen** in α: Since $i \in \{0 \ldots, m-1\}$ and all values $\{0, \ldots, m-1\}$ are decided in α by processes other than p_i, the value i is decided in α by some process p_j, $j \neq i$. But p_i is unseen in α, so p_j must have decided on i before p_i introduced i as an input value. Considering the same execution without the steps of p_i at the end, we conclude that the fact that p_j decides on i contradicts the validity property of the algorithm. Hence, there are no such tuples in X_{m+1}.
3. $\boldsymbol{i = m}$ and p_m is **unseen** in α: We show a bijection between this subset of X_{m+1} and C_m. Since p_m is unseen in α, in the beginning of α all processes but p_m take steps and decide on all values $0, \ldots, m-1$, and then p_m takes steps alone. Consider the execution $\hat{\alpha}$ induced by the same blocks, but excluding the steps of p_m at the end, and note that $\hat{\alpha}$ is in C_m. On the other hand, every $\hat{\alpha}$ in C_m can be extended to an execution α by adding singleton steps of p_m the its end, (α, p_m) is in X_{m+1} and p_m is unseen in α.
   By the induction hypothesis, the size of C_m is odd, so the bijection implies that X_{m+1} has an odd number of tuples (α, p_m) in which p_m is unseen in α.

Since the sizes of C_{m+1} and X_{m+1} have the same parity, and the latter size is odd, then so is the former. □

Taking $m = n$, we get that the size of C_n is odd, and hence, nonzero, implying that there is an execution in which all n values are decided, contradicting the $(n-1)$-agreement property.

Theorem 1. *There is no wait-free algorithm solving the $(n-1)$-set agreement task in an asynchronous shared memory system with n processes.*

4 Impossibility of Symmetry Breaking (SSB and WSB)

In weak symmetry breaking (WSB), n inputless processes should each output a single bit, satisfying:

Symmetry Breaking: If all processes output, then not all of them output the same value.

WSB is easily solvable when using the processes' identifiers: $p_0, \dots, p_{n-2}$ output 1, and p_{n-1} outputs 0. To prevent such solutions, we assume processes does not use their identifiers, and they all run the same algorithm. This restriction is enforced by demanding that WSB algorithm is *symmetric* [5,18,29], formalized as follows.

Let α be an execution prefix induced by a sequence of blocks $B_1 \cdots B_h$, and let $\pi : \{0, \dots, n-1\} \to \{0, \dots, n-1\}$ be a permutation. For a block B_j, let $\pi(B_j)$ be the block $\left\{p_{\pi(i)}\right\}_{p_i \in B_j}$, and denote by $\pi(\alpha)$ the execution prefix induced by $\pi(B_1) \cdots \pi(B_h)$.

A permutation $\pi : \{0, \dots, n-1\} \to \{0, \dots, n-1\}$ is *order preserving* on a set of processes P, if for every $p_i, p_{i'} \in P$, if $i < i'$ then $\pi(i) < \pi(i')$.

Definition 1. *An algorithm A is* symmetric *if, for every execution prefix α of A by a set of processes P, and for every permutation $\pi : \{0, \dots, n-1\} \to \{0, \dots, n-1\}$ that is order preserving on P, if a process p_i decides in α, then $p_{\pi(i)}$ decides in $\pi(\alpha)$, and on the same value.*

An adaptive version of WSB is the strong symmetry breaking (SSB) task, where n inputless processes should each output a single bit, satisfying the symmetry breaking property, and the following property:

1-decision: In every execution, at least one participating process outputs 1.

SSB is unsolvable even if the processes are allowed to use their identifiers, so we do not assume that algorithms are symmetric.

We can now explain the arguments used to prove the impossibility of both symmetry breaking tasks. As in the set agreement proof, we analyze the set of executions using counting arguments. Assume, towards a contradiction, that there is an algorithm A solving the relevant task—SSB or WSB. We associate A with a *univalued signed count*, a quantity that counts the executions of A in which all processes output the same value; if the univalued signed count is nonzero, then there is an illegal execution of A. We prove that for SSB, the univalued signed count is always nonzero, whereas for WSB, it is nonzero if n is a prime power.

To show that the univalued signed count is nonzero, we derive a trimmed version of A, with the same univalued signed count. Counting the univalued signed count is easier in the trimmed version: for SSB, it is immediate from the 1-decision property, and for WSB, the symmetric nature of the algorithm implies that the same values are output in different partial executions. Together with the fact that n is a prime power, this is used to show that the univalued signed count of the trimmed algorithm is nonzero, which completes the proof.

Section 4.1 defines the *sign* of an execution and presents a way to measure the size of a set of executions. Section 4.2 shows how to trim an algorithm in a way the preserves the univalued signed count. These tools are used in Sections 4.3 and 4.4 to prove the impossibility results for SSB and WSB, respectively.

4.1 Counting Executions by Signs

The *sign of a sequence of blocks* $B_1 \cdots B_h$ is $\text{sign}(B_1 \cdots B_h) = \prod_{i=1}^{h} (-1)^{|B_i|+1}$; it is positive if and only if there is an even number of even-sized blocks. The definition is crafted to obtain Proposition 1 and Lemma 3.

The *sign of an execution* α induced by $B_1 \cdots B_h$ is $\text{sign}(\alpha) = \text{sign}(B_1 \cdots B_h)$. If two executions (possibly of different algorithms) differ only in singleton steps of a process at their end, then the difference is only in odd-sized blocks, which do not change the sign, implying their signs are equal:

Proposition 1. *If α is an execution induced by $B_1 \cdots B_h$ and $\hat{\alpha}$ is an execution induced by $B_1 \cdots B_h \{p_i\}^m$, then* $\text{sign}(\alpha) = \text{sign}(\hat{\alpha})$.

Since the indistinguishable execution constructed in the proof of Lemma 1 is created by either pulling out or merging a singleton set, it must have an opposite sign. This is stated in the next lemma, extending Lemma 1 to argue about signs.

Lemma 3. *Let P be a set of processes, and let $p_i \in P$. If p_i is seen in an execution α of P, then there is a unique execution α' of P such that $\alpha' \overset{P-p_i}{\sim} \alpha$ and $\alpha' \neq \alpha$. Moreover, p_i is seen in α', and* $\text{sign}(\alpha') = -\text{sign}(\alpha)$.

This lemma is used in an analogous way to the parity argument in the proof of Lemma 2, except that here, we sum signs instead of checking the parity of a size; as in Lemma 2, pairs of executions from Lemma 3 cancel each other.

From now on, we consider only executions by all processes. For an algorithm A and for $v = 0, 1$, the set of executions of A in which only v is decided is $C_v^A = \{\alpha \mid dec(\alpha) = \{v\}\}$. These sets are defined for any algorithm, but the symmetry breaking property implies that both sets are empty, since the executions in which all processes decide on the same value, either 0 or 1, are prohibited. To prove that no algorithm solves SSB, or solves WSB when n is a prime power, we measure C_0^A and C_1^A and show they cannot both be empty.

The *signed count* of a set of executions S is $\mu(S) = \sum_{\alpha \in S} \text{sign}(\alpha)$. Clearly, $\mu(\emptyset) = 0$, and for any two disjoint sets S, T, $\mu(S \dot{\cup} T) = \mu(S) + \mu(T)$.

The *univalued signed count* of an algorithm A is $\mu\left(C_0^A\right) + (-1)^{n-1} \cdot \mu\left(C_1^A\right)$. Note that if the univalued signed count is nonzero, then A has an execution with a single output value. (The converse is not necessarily true, but this does not matter for the impossibility result.) We next define a trimmed version of A, $\text{T}(A)$, and show how this way of measuring C_0^A and C_1^A allows to prove that the univalued signed counts of A and $\text{T}(A)$ are equal (Lemma 4).

4.2 A Trimmed Algorithm

Let A be a wait-free algorithm which produces binary outputs. By assumption, process p_i alternates between write and scan operations:

```
write($i$) to $R_i$
while (1) do
    $\boldsymbol{v} \leftarrow$ Scan $(R_0, \ldots, R_{n-1})$
    Local$_A(\boldsymbol{v})$: [  calculation on $\boldsymbol{v}$
                                 if $cond$ then return $x$
                                 else write($\boldsymbol{v}$) to $R_i$ ]
```

We derive from A a *trimmed* algorithm, $\mathrm{T}(A)$, that does not claim to solve any symmetry breaking task. The code of $\mathrm{T}(A)$ for a process p_i is:

```
boolean $simulated = 0$
write($i$) to $R_i$
while (1) do
    $\boldsymbol{v} \leftarrow$ Scan $(R_0, \ldots, R_{n-1})$
    if $\boldsymbol{v}$ contains all processes then return $simulated$
    compute Local$_A(\boldsymbol{v})$
    if $A$ returns $x$ then return the same value $x$
    $simulated \leftarrow 1$
```

In every execution of $\mathrm{T}(A)$, the last process to take a first step sees all other processes in its first scan, takes no simulation steps and outputs 0; hence, $C_1^{\mathrm{T}(A)} = \emptyset$.

Every execution of A with an unseen process p_i is also an execution of $\mathrm{T}(A)$, up to the number of singleton steps of p_i at the end of the execution. By Proposition 1, changing the number of singleton steps does not affect the sign, so counting executions with an unseen process by sign is the same for both algorithms. This is used in the proof of the next lemma:

Lemma 4. *A and $\mathrm{T}(A)$ have the same univalued signed count.*

Proof. For each of the algorithms, we define an intermediate set of tuples in a way similar to the one used in the proof of the $(n-1)$-set agreement impossibility result (Lemma 1). These tuples contain executions spanning from the univalued 0 executions to the univalued 1 executions. More precisely, consider tuples of the form (α, p_i) such that in α, all processes with identifier smaller than i decide on 1, and all processes with identifier greater than i decide on 0. As in the proof of Lemma 1, the output of p_i does not matter. For an algorithm A, let:

$$X^A = \{(\alpha, p_i) \mid dec(\alpha, \{p_0, \ldots, p_{i-1}\}) = \{1\}; dec(\alpha, \{p_{i+1}, \ldots, p_{n-1}\}) = \{0\}\}.$$

We abuse notation and define the *signed count* of the set of pairs X^A to be $\lambda\left(X^A\right) = \sum_{i=0}^{n-1} (-1)^i \mu(\{\alpha \mid (\alpha, p_i) \in X^A\})$. Again, $\lambda(\emptyset) = 0$, and for any two disjoint sets S, T, $\lambda(S \dot\cup T) = \lambda(S) + \lambda(T)$. Similar notations are used for $\mathrm{T}(A)$.

The $(-1)^i$ element in $\lambda\left(X^A\right)$ is used to cancel out pairs of tuples in X^A with the same execution and processes with consecutive identifiers. This is used in the first case of the next claim:

Claim. The univalued signed count of an algorithm A equals $\lambda\left(X^A\right)$.

Proof of claim. For every tuple $(\alpha, p_i) \in X^A$ there are three possibilities:

1. $dec(\alpha) = \{0,1\}$. If $dec(\alpha, p_i) = 1$ then
$$dec(\alpha, \{p_0, \ldots, p_{i-1}, p_i\}) = \{1\};\ dec(\alpha, \{p_{i+1}, \ldots, p_{n-1}\}) = \{0\},$$
so (α, p_{i+1}) is also in X^A. In $\lambda\left(X^A\right)$, α appears exactly twice, for (α, p_i) and for (α, p_{i+1}), and the corresponding summands cancel each other, since $(-1)^i \operatorname{sign}(\alpha) = -(-1)^{i+1} \operatorname{sign}(\alpha)$.
If $dec(\alpha, p_i) = 0$ then, by a similar argument, (α, p_{i-1}) is also in X^A and the appropriate summands cancel each other.
2. $dec(\alpha) = \{0\}$. Then $i = 0$, α appears in X^A once, as (α, p_0), and its sign in $\lambda\left(X^A\right)$ is $\operatorname{sign}(\alpha)$, as in $\mu\left(C_0^A\right)$.
3. $dec(\alpha) = \{1\}$. Then $i = n-1$, α appears in X^A once, as (α, p_{n-1}), and its sign in $\lambda\left(X^A\right)$ is $(-1)^{n-1} \operatorname{sign}(\alpha)$, as in $(-1)^{n-1}\mu\left(C_1^A\right)$.

Therefore, every tuple in X^A implies either two summands in $\lambda\left(X^A\right)$ that cancel each other, or a summand that appears in $\lambda\left(X^A\right)$ and in $\mu\left(C_0^A\right) + (-1)^{n-1}\mu\left(C_1^A\right)$ with the same sign. On the other hand, every execution $\alpha \in C_0^A$ appears in X^A in a pair (α, p_0), as discussed in the second case, and every $\alpha \in C_1^A$ appears in X^A in a pair (α, p_{n-1}), as discussed in the third case. Hence, the sums are equal. □

It remains to show that $\lambda\left(X^A\right) = \lambda\left(X^{\mathrm{T}(A)}\right)$.

For a tuple $(\alpha, p_i) \in X^A$ such that p_i is unseen in α, consider the execution $\bar{\alpha}$ of $\mathrm{T}(A)$ with the same sequence of blocks, possibly omitting singleton steps at the end; by Proposition 1, both executions have the same sign. Moreover, all processes but p_i complete the simulation of A and output the same values as in α. Hence, $(\bar{\alpha}, p_i) \in X^{\mathrm{T}(A)}$ and the contribution of (α, p_i) to $\lambda\left(X^A\right)$ equals the contribution of $(\bar{\alpha}, p_i)$ to $\lambda\left(X^{\mathrm{T}(A)}\right)$.

The sum over tuples (α, p_i) in which p_i is seen in α is 0 in both cases: fix a process p_i, and consider all the tuples $(\alpha, p_i) \in X^A$ in which p_i is seen in α. By Lemma 3, for each α there is a unique execution $\alpha' \neq \alpha$ of A such that $\alpha \overset{\mathbb{P}-p_i}{\sim} \alpha'$, hence for each $p_j \neq p_i$, $dec(\alpha, p_j) = dec(\alpha', p_j)$, and $(\alpha', p_i) \in X^A$. Moreover, p_i is also seen in α', and $\operatorname{sign}(\alpha) = -\operatorname{sign}(\alpha')$. Hence, we can divide all these tuples into pairs, (α, p_i) and (α', p_i), so that $\operatorname{sign}(\alpha) = -\operatorname{sign}(\alpha')$, each of which cancels out in $\lambda\left(X^A\right)$. The same claim holds for $\mathrm{T}(A)$ as well. □

4.3 Impossibility of Strong Symmetry Breaking

Let S be an SSB algorithm and consider its trimmed version, $\mathrm{T}(S)$. By Lemma 4, the univalued signed count of S and $\mathrm{T}(S)$ is the same, so it suffices to prove:

Lemma 5. *The univalued signed count of* $\mathrm{T}(S)$ *is nonzero.*

Proof. In every execution of $\mathrm{T}(S)$, the last process to take a first step sees all other processes in its first step and decides on *simulated* $= 0$, hence $C_1^{\mathrm{T}(S)} = \emptyset$.

There is a unique execution α of $\mathrm{T}(S)$ in which all processes take their first step together, take no simulation steps and decide 0. So $\alpha \in C_0^{\mathrm{T}(S)}$, and we claim there is no other execution in $C_0^{\mathrm{T}(S)}$. Consider another execution α' of $\mathrm{T}(S)$. In α', there is a set of processes which does not see all other processes in their first scan operation, and all of them set *simulated* $= 1$. If one of these processes sees all other processes in a later scan, it decides 1. Otherwise, all these processes decide in the simulation of S, and this simulation induces a legal execution of S. By the 1-decision property of S, at least one of them decides 1 in S, and hence in $\mathrm{T}(S)$.

Hence, $C_0^{\mathrm{T}(S)} = \{\alpha\}$, and since $C_1^{\mathrm{T}(S)} = \emptyset$, the univalued signed count of $\mathrm{T}(S)$ is $\mathrm{sign}(\alpha) = (-1)^{n+1}$. □

By Lemma 4, the univalued signed count of S is also nonzero. Hence, there is an execution of S where all processes decide, and on the same value, so the algorithm does not satisfy the symmetry breaking property.

Theorem 2. *There is no wait-free algorithm solving SSB in an asynchronous shared memory system with any number of processes.*

4.4 Impossibility of Weak Symmetry Breaking

Let W be a symmetric WSB algorithm, and consider its trimmed version $\mathrm{T}(W)$.

In order to compute the univalued signed count of $\mathrm{T}(W)$, we use the fact that W is symmetric to show that every execution α of $\mathrm{T}(W)$ where not all processes participate has a set of executions with the same outputs as in α. This is formalized by defining an equivalence relation on the executions of $\mathrm{T}(W)$ and considering the equivalence classes it induces.

For an execution α of $\mathrm{T}(W)$, induced by the blocks $B_1 \cdots B_h$, let ℓ be the index of the first block after all processes have taken a step, i.e., $\cup_{j \leq \ell} B_j = \mathbb{P}$ and $\cup_{j<\ell} B_j \neq \mathbb{P}$. P_α is the set of all processes that set the simulated variable to be 1, namely, took steps before the rest of the processes appear. Formally, $P_\alpha = \cup_{j<\ell} B_j$, and $P_\alpha = \emptyset$ if $B_1 = \mathbb{P}$; denote $\overline{P_\alpha} = \mathbb{P} \setminus P_\alpha$.

For an integer m, $0 \leq m < n$, let S_m be the set of executions α such that $|P_\alpha| = m$, i.e., in which exactly m processes take steps in W. Note that every execution α is in some set S_m, since P_α is defined for every α, and $|P_\alpha| < n$ since the last process to start does not simulate a step of W.

When $m = 0$, S_0 is the set of executions in which all processes take steps in the first block. In $\mathrm{T}(W)$, they all decide 0 and halt, so S_0 contains only the execution induced by the single block $\{p_0, \ldots, p_{n-1}\}$. Hence, $|S_0| = 1$.

Fix some $m \geq 0$ and consider two executions $\alpha, \alpha' \in S_m$, such that α is induced by $B_1 \cdots B_h$ and α' is induced by $B'_1 \cdots B'_{h'}$. Define a relation $\simeq$ on S_m, such that $\alpha \simeq \alpha'$ if and only if $h = h'$, and there is a permutation $\pi : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ that is order preserving on P_α and on $\overline{P_\alpha}$, such that $B'_j = \pi(B_j)$ for every $j \in \{1, \ldots, h\}$. It is easy to check that $\simeq$ is an equivalence relation. The *equivalence class* of an execution α is $[\alpha] = \{\alpha' \mid \alpha \simeq \alpha'\}$.

Since W is symmetric, it can be verified that the equivalence classes of $\simeq$ have the following useful properties:

Proposition 2. *If $\alpha \in S_m$ and $\alpha' \simeq \alpha$, then* $\mathrm{sign}(\alpha) = \mathrm{sign}(\alpha')$*,* $\alpha' \in S_m$*, and* $dec(\alpha) = dec(\alpha')$*.*

Therefore, we can denote the sign of all the executions in $[\alpha]$ by $\mathrm{sign}([\alpha])$.

For two sets of equal sizes, P and P', note that there is a unique permutation $\pi : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ that maps P to P' and $\mathbb{P} \setminus P$ to $\mathbb{P} \setminus P'$ and is order preserving on P and on $\mathbb{P} \setminus P$. This is due to the fact that all identifiers are distinct and hence, there is a single way to map P to P' in an order-preserving manner, and a single way to map $\mathbb{P} \setminus P$ to $\mathbb{P} \setminus P'$ in such manner. This implies that π is unique, which is used in the proof of the next lemma:

Lemma 6. *For every m, $0 \le m < n$, and for every execution $\alpha \in S_m$, the size of $[\alpha]$ is $\binom{n}{m}$.*

Proof. Denote by $\binom{\mathbb{P}}{m}$ the set of all subsets of $\mathbb{P}$ of size m. Then $\left|\binom{\mathbb{P}}{m}\right| = \binom{n}{m}$. For $\alpha \in S_m$, define $f : [\alpha] \to \binom{\mathbb{P}}{m}$ by $f(\alpha') = P_{\alpha'}$, which is well defined by Proposition 2. We prove that f is a bijection.

Let $\alpha', \alpha'' \in [\alpha]$ satisfying $f(\alpha') = f(\alpha'')$, and assume $\alpha' = \pi(\alpha)$ and $\alpha'' = \varphi(\alpha)$, for two permutations π, φ that are order preserving on P_α and on $\overline{P_\alpha}$. Since $f(\alpha') = f(\alpha'')$, we have that $P_{\alpha'} = P_{\alpha''}$, and hence, $\overline{P_{\alpha'}} = \overline{P_{\alpha''}}$. Since there is a unique permutation that maps P_α to $P_{\alpha'}$ and $\overline{P_\alpha}$ to $\overline{P_{\alpha'}}$ in an order-preserving manner, it follows that $\pi = \varphi$, implying that $\alpha' = \alpha''$.

Let $P \in \binom{\mathbb{P}}{m}$, and denote by π the unique permutation that maps P_α to P and $\overline{P_\alpha}$ to $\mathbb{P} \setminus P$ and is order preserving on both sets. Let $\alpha' = \pi(\alpha)$, so $\alpha' \in S_m$ and $f(\alpha') = P_{\alpha'} = P$. □

Lemma 7. *If $n = q^e$ for a prime number q and a positive integer e, then the univalued signed count of* $\mathrm{T}(W)$ *is nonzero.*

Proof. In every execution of $\mathrm{T}(W)$, at least one process sees all other processes in its first scan and decides 0. Therefore, $C_1^{\mathrm{T}(W)} = \emptyset$ and $\mu\left(C_1^{\mathrm{T}(W)}\right) = 0$, implying that the univalued signed count of $\mathrm{T}(W)$ is equal to $\mu\left(C_0^{\mathrm{T}(W)}\right)$. To show that $\mu\left(C_0^{\mathrm{T}(W)}\right)$ is nonzero, note that $C_0^{\mathrm{T}(W)}$ is the disjoint union of $C_0^{\mathrm{T}(W)} \cap S_m$ for $0 \le m \le n-1$, since each execution is in some set S_m. Hence,

$$\mu\left(C_0^{\mathrm{T}(W)}\right) = \sum_{m=0}^{n-1} \mu\left(C_0^{\mathrm{T}(W)} \cap S_m\right).$$

Fix $m \ge 0$. By Proposition 2, if α is in $C_0^{\mathrm{T}(W)} \cap S_m$ then every $\alpha' \simeq \alpha$ is also in $C_0^{\mathrm{T}(W)} \cap S_m$. Therefore, $C_0^{\mathrm{T}(W)} \cap S_m$ consists of complete equivalence classes, and can be rewritten as $\cup_{\alpha \in C_0^{\mathrm{T}(W)} \cap S_m} [\alpha]$. Therefore,

$$\mu\left(C_0^{\mathrm{T}(W)}\right) = \sum_{m=0}^{n-1} \sum_{\{[\alpha] \mid \alpha \in C_0^{\mathrm{T}(W)} \cap S_m\}} \mu\left([\alpha]\right).$$

By Lemma 6, the size of each equivalence class is $\binom{n}{m}$. Also, recall that all executions in an equivalence class have the same sign. Note that S_0 contains only the execution α induced by the block $\{p_0, \ldots, p_{n-1}\}$; for this execution, $P_\alpha = \emptyset$, and its sign is $(-1)^{n+1}$. Therefore,

$$\mu\left(C_0^{\mathrm{T}(W)}\right) = (-1)^{n+1} + \sum_{m=1}^{n-1} \sum_{\{[\alpha] \mid \alpha \in C_0^{\mathrm{T}(W)} \cap S_m\}} \binom{n}{m} \cdot \mathrm{sign}([\alpha]).$$

The following basic result of number theory follows from Lucas' Theorem.

Claim. If q is a prime number and e, m are positive integers such that $0 < m < q^e$, then $\binom{q^e}{m} \equiv 0 \pmod{q}$.

Therefore, all summands, except the first one, are 0 mod q, and hence,

$$\mu\left(C_0^{\mathrm{T}(W)}\right) \equiv (-1)^{n+1} \not\equiv 0 \pmod{q},$$

hence, $\mu\left(C_0^{\mathrm{T}(W)}\right) \neq 0$ and the univalued signed count of $\mathrm{T}(W)$ is nonzero. $\square$

Consider a WSB algorithm W for n processes, where n is a prime power. By Lemma 7, the univalued signed count of $\mathrm{T}(W)$ is nonzero, and by Lemma 4, the same holds for W. This implies that at least one of C_0^W and C_1^W is not empty. Thus, there is an execution of W in which only 0 or only 1 is decided, contradicting the symmetry breaking property of W.

Theorem 3. *There is no wait-free algorithm solving WSB in an asynchronous shared memory system if the number of processes is a prime power.*

5 Discussion

Understanding wait-free solvable tasks is at the heart of distributed computing. This paper suggests a new approach for studying wait-free solvability of sub-consensus tasks. The novel ingredient in our approach is in *counting* (sometimes by sign) the number of executions that violate the task's requirements, as a way to show that such executions exist. This yields simple impossibility proofs for wait-free computation using only read and write operations: n processes cannot solve $(n-1)$-set agreement or SSB, and, if n is a prime power, they cannot solve $(2n-2)$-renaming. The simplicity of the proofs, and in particular, the fact they use only elementary mathematics, should make them accessible to a wider audience and increase confidence in their correctness. We hope this better understanding of colored tasks will promote further investigation of them.

Prior approaches [3, 5, 12, 28] also consider a restricted subset of executions in which all processes decide. Additionally, some parts of our proofs are analogous to known topological proofs [13, 28]. In these proofs, simplexes represent executions, where each node represents the local view of a process. These papers use variants of Lemma 1 to prove that the set of simplexes representing block executions induces a *manifold*.

An execution α with the view of a process p_i factored out is represented by the pair (α, p_i) in our proofs; this is the analogue of the *face* of the simplex of α that is opposite to the node of p_i. Thus, counting pairs is analogous to counting faces of a simplex, as used in a classical proof of Sperner's Lemma, and in the proof of the Index lemma [24]. Indeed, Lemma 2 is analogous to Sperner's Lemma, and Lemma 4 corresponds to the use of the Index lemma in [12,13].

The sign of an execution is used here instead of the topological notion of defining an *orientation* on a manifold and comparing it with the orientation of each simplex; univalued signed count is the counterpart of the topological *content*. The trimmed algorithm $\mathrm{T}(W)$ is an algorithmic way to look at the *cone construction* of [13]. Proposition 2 is proved there in a relatively complicated manner, using i-corners or *flip* operations and paths in a subdivided simplex.

Simulations, like [7,17,20], can be used to translate the lower bounds to other models, e.g., message-passing systems or models with fewer failures. More interestingly, it should be possible to derive analogous lower bounds for other models, by showing properties similar to Lemma 3. For example, in the message-passing model, we can consider layered executions [30] and take the sign according to the number of messages sent in the layers.

Finally, and perhaps most importantly, our simpler proofs may lead to the discovery of matching upper bounds, for example, an explicit $(2n-2)$-renaming algorithm when n is not a prime power.

Acknowledgements. We would like to thank Maurice Herlihy for collaboration in earlier stages of this research, Armando Castañeda, Keren Censor-Hillel, Faith Ellen, Tal Horesh, Petr Kuznetsov, Sergio Rajsbaum, Mor Weiss and the referees for valuable comments, and Amir Shpilka for discussions on number theory.

References

1. Attiya, H.: A direct lower bound for k-set consensus. In: PODC 1998, p. 314 (1998)
2. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. J. ACM 37, 524–548 (1990)
3. Attiya, H., Castañeda, A.: A Non-topological Proof for the Impossibility of k-Set Agreement. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 108–119. Springer, Heidelberg (2011)
4. 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)
5. Attiya, H., Rajsbaum, S.: The combinatorial structure of wait-free solvable tasks. SIAM J. Comput. 31, 1286–1313 (2002)
6. Attiya, H., Welch, J.: Distributed computing: fundamentals, simulations, and advanced topics. Wiley series on parallel and distributed computing. Wiley (2004)
7. Borowsky, E., Gafni, E., Lynch, N., Rajsbaum, S.: The BG distributed simulation algorithm. Distributed Computing 14, 127–146 (2001)
8. Borowsky, E., Gafni, E.: Generalized FLP impossibility result for t-resilient asynchronous computations. In: STOC 1993, pp. 91–100 (1993)

9. Borowsky, E., Gafni, E.: Immediate atomic snapshots and fast renaming. In: PODC 1993, pp. 41–51 (1993)
10. Castañeda, A.: A Study of the Wait-free Solvability of Weak Symmetry Breaking and Renaming. PhD thesis, Universidad Nacional Autonoma de Mexico (2010)
11. Castañeda, A., Herlihy, M., Rajsbaum, S.: An Equivariance Theorem with Applications to Renaming. In: Fernández-Baca, D. (ed.) LATIN 2012. LNCS, vol. 7256, pp. 133–144. Springer, Heidelberg (2012)
12. Castañeda, A., Rajsbaum, S.: New combinatorial topology upper and lower bounds for renaming. In: PODC 2008, pp. 295–304 (2008)
13. Castañeda, A., Rajsbaum, S.: New combinatorial topology bounds for renaming: the lower bound. Distributed Computing 22, 287–301 (2010)
14. Castañeda, A., Rajsbaum, S., Raynal, M.: The renaming problem in shared memory systems: An introduction. Computer Science Review 5(3), 229–251 (2011)
15. Castañeda, A., Rajsbaum, S.: New combinatorial topology bounds for renaming: the upper bound. J. ACM 59(1) (2012)
16. Chaudhuri, S.: More choices allow more faults: set consensus problems in totally asynchronous systems. Inf. Comput. 105(1), 132–158 (1993)
17. Gafni, E.: Round-by-round fault detectors: Unifying synchrony and asynchrony. In: PODC 1998, pp. 143–152 (1998)
18. Gafni, E.: Read-write reductions. In: ICDCN 2006, pp. 349–354 (2006)
19. 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)
20. Gafni, E.: The extended BG-simulation and the characterization of t-resiliency. In: STOC 2009, pp. 85–92 (2009)
21. Gafni, E., Rajsbaum, S.: Recursion in Distributed Computing. In: Dolev, S., Cobb, J., Fischer, M., Yung, M. (eds.) SSS 2010. LNCS, vol. 6366, pp. 362–376. Springer, Heidelberg (2010)
22. Gafni, E., Rajsbaum, S., Herlihy, M.P.: Subconsensus Tasks: Renaming Is Weaker Than Set Agreement. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 329–338. Springer, Heidelberg (2006)
23. Hardy, G.: Ramanujan: Twelve Lectures on Subjects Suggested by His Life and Work. AMS Chelsea Publishing Series. AMS Chelsea Pub. (1999)
24. Henle, M.: A Combinatorial Introduction to Topology. Dover Books on Mathematics Series. Dover (1994)
25. Herlihy, M.: Impossibility results for asynchronous PRAM. In: SPAA 1991, pp. 327–336 (1991)
26. Herlihy, M., Rajsbaum, S.: Algebraic spans. Math. Struct. Comp. Sci. 10, 549–573 (2000)
27. Herlihy, M., Shavit, N.: The asynchronous computability theorem for t-resilient tasks. In: STOC 1993, pp. 111–120 (1993)
28. Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. J. ACM 46, 858–923 (1999)
29. Imbs, D., Rajsbaum, S., Raynal, M.: The Universe of Symmetry Breaking Tasks. In: Kosowski, A., Yamashita, M. (eds.) SIROCCO 2011. LNCS, vol. 6796, pp. 66–77. Springer, Heidelberg (2011)
30. Moses, Y., Rajsbaum, S.: A layered analysis of consensus. SIAM J. Comput. 31(4), 989–1021 (2002)
31. Raynal, M., Travers, C.: Synchronous set agreement: a concise guided tour. In: PRDC 2006, pp. 267–274 (2006)
32. Saks, M., Zaharoglou, F.: Wait-free k-set agreement is impossible: The topology of public knowledge. SIAM J. Comput. 29(5), 1449–1483 (2000)

Randomized Distributed Decision

Pierre Fraigniaud[1], Amos Korman[1,⋆], Merav Parter[2], and David Peleg[2,⋆⋆]

[1] CNRS and University Paris Diderot, France
{pierre.fraigniaud,amos.korman}@liafa.jussieu.fr
[2] The Weizmann Institute of Science, Rehovot, Israel
{merav.parter,david.peleg}@weizmann.ac.il

Abstract. The paper tackles the power of randomization in the context of locality by analyzing the ability to "boost" the success probability of deciding a distributed language. The main outcome of this analysis is that the distributed computing setting contrasts significantly with the sequential one as far as randomization is concerned. Indeed, we prove that in some cases, the ability to increase the success probability for deciding distributed languages is rather limited.

We focus on the notion of a (p, q)-*decider* for a language $\mathcal{L}$, which is a distributed randomized algorithm that *accepts* instances in $\mathcal{L}$ with probability at least p and *rejects* instances outside of $\mathcal{L}$ with probability at least q. It is known that every hereditary language that can be decided in t rounds by a (p, q)-decider, where $p^2 + q > 1$, can be decided *deterministically* in $O(t)$ rounds. One of our results gives evidence supporting the conjecture that the above statement holds for all distributed languages and not only for hereditary ones, by proving the conjecture for the restricted case of path topologies.

For the range below the aforementioned threshold, namely, $p^2 + q \leq 1$, we study the class $B_k(t)$ (for $k \in \mathbb{N}^* \cup \{\infty\}$) of all languages decidable in at most t rounds by a (p, q)-decider, where $p^{1+\frac{1}{k}} + q > 1$. Since every language is decidable (in zero rounds) by a (p, q)-decider satisfying $p + q = 1$, the hierarchy B_k provides a spectrum of complexity classes between determinism ($k = 1$, under the above conjecture) and complete randomization ($k = \infty$). We prove that all these classes are separated, in a strong sense: for every integer $k \geq 1$, there exists a language $\mathcal{L}$ satisfying $\mathcal{L} \in B_{k+1}(0)$ but $\mathcal{L} \notin B_k(t)$ for any $t = o(n)$. In addition, we show that $B_\infty(t)$ does not contain all languages, for any $t = o(n)$. In other words, we obtain the hierarchy $B_1(t) \subset B_2(t) \subset \cdots \subset B_\infty(t) \subset$ All.

Finally, we show that if the inputs can be restricted in certain ways, then the ability to boost the success probability becomes almost null, and in particular, derandomization is not possible even beyond the threshold $p^2 + q = 1$.

Keywords: Local distributed algorithms, local decision, randomized algorithms.

⋆ Supported by the ANR projects DISPLEXITY and PROSE, and by the INRIA project GANG.

⋆⋆ Supported in part by the Israel Science Foundation (grant 894/09), the US-Israel Binational Science Foundation (grant 2008348), the Israel Ministry of Science and Technology (infrastructures grant), and the Citi Foundation.

1 Introduction

Background and Motivation. The impact of randomization on computation is one of the most central questions in computer science. In particular, in the context of distributed computing, the question of whether randomization helps in improving locality for construction problems has been studied extensively. While most of these studies were problem-specific, several attempts have been made for tackling this question from a more general and unified perspective. For example, Naor and Stockmeyer [26] focus on a class of problems called LCL (essentially a subclass of the class LD discussed below), and show that if there exists a randomized algorithm that constructs a solution for a problem in LCL in a constant number of rounds, then there is also a constant time deterministic algorithm constructing a solution for that problem.

Recently, this question has been studied in the context of *local decision*, where one aims at deciding locally whether a given global input instance belongs to some specified language [13]. The localities of deterministic algorithms and randomized Monte Carlo algorithms are compared in [13], in the $\mathcal{LOCAL}$ model (cf. [28]). One of the main results of [13] is that randomization does not help for locally deciding *hereditary* languages if the success probability is beyond a certain guarantee threshold. More specifically, a (p, q)*-decider* for a language $\mathcal{L}$ is a distributed randomized Monte Carlo algorithm that *accepts* instances in $\mathcal{L}$ with probability at least p and *rejects* instances outside of $\mathcal{L}$ with probability at least q. It was shown in [13] that every hereditary language that can be decided in t rounds by a (p, q)-decider, where $p^2 + q > 1$, can actually be decided *deterministically* in $O(t)$ rounds. On the other hand, [13] showed that the aforementioned threshold is sharp, at least when hereditary languages are concerned. In particular, for every p and q, where $p^2 + q \leq 1$, there exists an hereditary language that cannot be decided deterministically in $o(n)$ rounds, but can be decided in zero rounds by a (p, q)-decider.

In one of our results we provide evidence supporting the conjecture that the above statement holds for all distributed languages and not only for hereditary ones. This is achieved by considering the restricted case of path topologies. In addition, we present a more refined analysis for the family of languages that can be decided randomly but not deterministically. That is, we focus on the family of languages that can be decided locally by a (p, q)-decider, where $p^2 + q \leq 1$, and introduce an infinite hierarchy of classes within this family, characterized by the specific relationships between the parameters p and q. As we shall see, our results imply that the distributed computing setting contrasts significantly with the sequential one as far as randomization is concerned. Indeed, we prove that in some cases, the ability to increase the success probability for deciding distributed languages is very limited.

Model. We consider the $\mathcal{LOCAL}$ model (cf. [28]), which is a standard distributed computing model capturing the essence of spatial locality. In this model, processors are woken up simultaneously, and computation proceeds in fault-free synchronous rounds during which every processor exchanges messages of unlimited size with its neighbors, and performs arbitrary computations on its data. It is important to stress that all the algorithmic constructions that we employ in our positive results use messages of constant size (some of which do not use any communication at all). Hence, all our results apply not only to the $\mathcal{LOCAL}$ model of computation but also to more restricted models, for example, the $\mathcal{CONGEST}(B)$ model[1], where $B = O(1)$.

A distributed algorithm $\mathcal{A}$ that runs on a graph G operates separately on each connected component, and nodes of a component C of G cannot distinguish the underlying graph G from C. Therefore, we consider connected graphs only.

We focus on *distributed decision tasks.* Such a task is characterized by a finite or infinite set Σ of symbols (e.g., $\Sigma = \{0, 1\}$, or $\Sigma = \{0, 1\}^*$), and by a *distributed language* $\mathcal{L}$ defined on this set of symbols (see below). An *instance* of a distributed decision task is a pair $(G, \mathbf{x})$ where G is an n-node connected graph, and $\mathbf{x} \in \Sigma^n$, that is, every node $v \in V(G)$ is assigned as its *local input* a value $\mathbf{x}(v) \in \Sigma$. (In some cases, the local input of every node is empty, i.e., $\Sigma = \{\epsilon\}$, where ϵ denotes the empty binary string.) We define a *distributed language* as a decidable collection $\mathcal{L}$ of instances[2].

In the context of distributed computing, each processor must produce a boolean output, and the decision is defined by the conjunction of the processors outputs, i.e., if the instance belongs to the language, then all processors must output "yes", and otherwise, at least one processor must output "no". Formally, for a distributed language $\mathcal{L}$, we say that a distributed algorithm $\mathcal{A}$ *decides* $\mathcal{L}$ if and only if for every instance $(G, \mathbf{x})$ and id-assignment Id, every node v of G eventually terminates and produces an output denoted $\text{out}_{\mathcal{A}}(G, \mathbf{x}, \text{Id}, v)$, which is either "yes" or "no", satisfying the following decision rules:

- If $(G, \mathbf{x}) \in \mathcal{L}$, then $\text{out}_{\mathcal{A}}(G, \mathbf{x}, \text{Id}, v) =$ "yes" for every node $v \in V(G)$;
- If $(G, \mathbf{x}) \notin \mathcal{L}$, then $\text{out}_{\mathcal{A}}(G, \mathbf{x}, \text{Id}, v) =$ "no" for at least one node $v \in V(G)$.

Decision problems provide a natural framework for tackling fault-tolerance: the processors have to collectively check if the network is fault-free, and a node detecting a fault raises an alarm. Many natural problems can be phrased as decision problems, for example: "is the network planar?" or "is there a unique leader in the network?". Moreover, decision problems occur naturally when one aims at checking the validity of the output of a computational task, such as "is the produced coloring legal?", or "is the constructed subgraph an MST?".

The class of decision problems that can be solved in at most t communication rounds is denoted by $\text{LD}(t)$, for *local decision.* More precisely, let t be a function of triplets $(G, \mathbf{x}, \text{Id})$, where Id denotes the identity assignment to the nodes of G. Then $\text{LD}(t)$ is the class of all distributed languages that can be decided by a distributed algorithm that runs in at most t communication rounds. The randomized (Monte Carlo 2-sided error) version of the class $\text{LD}(t)$ is denoted $\text{BPLD}(t, p, q)$, which stands for *bounded-error probabilistic local decision*, and provides an analog of BPP for distributed computing, where p and q respectively denote the yes-error and the no-error guarantees. More precisely, a *randomized*

[1] The $\mathcal{CONGEST}(B)$ model is similar to the $\mathcal{LOCAL}$ model, except that message size is assumed to be bounded by B (cf. [28]).

[2] Note that an undecidable collection of instances remains undecidable in the distributed setting too.

distributed algorithm is a distributed algorithm $\mathcal{A}$ that enables every node v, at any round r during its execution, to generate a certain number of random bits. For constants $p, q \in (0, 1]$, we say that a randomized distributed algorithm $\mathcal{A}$ is a *(p, q)-decider* for $\mathcal{L}$, or, that it decides $\mathcal{L}$ with "yes" success probability p and "no" success probability q, if and only if for every instance $(G, \mathbf{x})$ and id-assignment Id, every node of G eventually terminates and outputs "yes" or "no", and the following properties are satisfied:

- If $(G, \mathbf{x}) \in \mathcal{L}$ then $\Pr[\forall v \in V(G),\ \text{out}_{\mathcal{A}}(G, \mathbf{x}, \text{Id}, v) = \text{"yes"}] \geq p$;
- If $(G, \mathbf{x}) \notin \mathcal{L}$ then $\Pr[\exists v \in V(G),\ \text{out}_{\mathcal{A}}(G, \mathbf{x}, \text{Id}, v) = \text{"no"}] \geq q$.

The probabilities in the above definition are taken over all possible coin tosses performed by the nodes. The running time of a (p, q)-decider executed on a node v depends on the triple $(G, \mathbf{x}, \text{Id})$ and on the results of the coin tosses. In the context of a randomized algorithm, $T_v(G, \mathbf{x}, \text{Id})$ denotes the maximal running time of the algorithm on v over all possible coin tosses, for the instance $(G, \mathbf{x})$ and id-assignment Id. Now, just as in the deterministic case, the running time T of the (p, q)-decider is the maximum running time over all nodes. Note that by definition of the distributed Monte-Carlo algorithm, both T_v and T are deterministic. For constant $p, q \in (0, 1]$ and a function t of triplets $(G, \mathbf{x}, \text{Id})$, $\text{BPLD}(t, p, q)$ is the class of all distributed languages that have a randomized distributed (p, q)-decider running in time at most t (i.e., can be decided in time at most t by a randomized distributed algorithm with "yes" success probability p and "no"success probability q).

Our main interest within this context is in studying the connections between the classes $\text{BPLD}(t, p, q)$. In particular, we are interested in the question of whether one can "boost" the success probabilities of a (p, q)-decider. (Recall that in the *sequential* Monte Carlo setting, such "boosting" can easily be achieved by repeating the execution of the algorithm a large number of times.) Our starting point is the recent result of [13] that, for the class of hereditary languages (i.e., closed under sub-graphs), the relation $p^2 + q = 1$ is a sharp threshold for randomization. That is, for hereditary languages, $\bigcup_{p^2+q>1} \text{BPLD}(t, p, q)$ collapses to $\text{LD}(O(t))$, but for any $p, q \in (0, 1]$ such that $p^2 + q \leq 1$ there exists a language $\mathcal{L} \in \text{BPLD}(0, p, q)$, while $\mathcal{L} \notin \text{LD}(t)$ for any $t = o(n)$. We conjecture that the hereditary assumption can be removed and we give some evidence supporting this conjecture. Aiming at analyzing the collection of classes $\bigcup_{p^2+q\leq 1} \text{BPLD}(t, p, q)$, we consider the set of classes

$$B_\infty(t) = \bigcup_{p+q>1} \text{BPLD}(t, p, q), \quad B_k(t) = \bigcup_{p^{1+1/k}+q>1} \text{BPLD}(t, p, q) \text{ for any } k \in \mathbb{Z}^+ .$$

Hence, our conjecture states that $B_1(t) = \text{LD}(O(t))$. Note that the class $\bigcup_{p+q\geq 1} \text{BPLD}(0, p, q)$ contains *all* languages, using a $(1, 0)$-decider that systematically returns "yes" at every node (without any communication). Hence, the classes B_k provide a smooth spectrum of randomized distributed complexity classes, from the class of deterministically decidable languages (under our conjecture) to the class of all languages. The ability of boosting the success probabilities of a (p, q)-decider is directly related to the question of whether these classes are different, and to what extent.

Our Results. One of the main outcomes of this paper is a proof that boosting success probabilities in the distributed setting appears to be quite limited. By definition, $B_k(t) \subseteq B_{k+1}(t)$ for any k and t. We prove that these inclusions are strict. In fact, we prove a stronger separation result: there exists a language in $B_{k+1}(0)$ that is not in $B_k(t)$ for any $t = o(n)$, and moreover, $\texttt{Tree} \notin B_\infty(t)$ for any $t = o(n)$, where $\texttt{Tree} = \{(G, \epsilon) : G \text{ is a tree}\}$. Hence $B_\infty(t)$ does not contain all languages, even for $t = o(n)$. In summary, we obtain the hierarchy

$$B_1(t) \subset B_2(t) \subset \cdots \subset B_\infty(t) \subset \text{All} .$$

These results demonstrate that boosting the probability of success might be doable, but only from a (p, q) pair satisfying $p^{1+1/(k+1)} + q > 1$ to a (p, q) pair satisfying $p^{1+1/k} + q > 1$ (with the extremes excluded). It is an open question whether $B_{k+1}(t)$ actually collapses to $\text{BPLD}(O(t), p, q)$, where $p^{1+1/k} + q = 1$, or whether there exist intermediate classes.

Recall that every hereditary language in $B_1(t)$ is also in $\text{LD}(O(t))$ [13]. We conjecture that this derandomization result holds for all languages and not only for hereditary ones. We give evidence supporting this conjecture by showing that restricted to path topologies, finite input and constant running time t, the statement $B_1(t) \subseteq \text{LD}(O(t))$ holds without assuming the hereditary property. This evidence seems to be quite meaningful especially since all our separation results hold even if we restrict ourselves to decision problems on path topologies.

Finally, we show that the situation changes drastically if the distribution of inputs can be restricted in certain ways. Indeed, we show that for every two reals $0 < r < r'$, there exists a language in $C_{r'}(0)$ that is not in $C_r(t)$ for any $t = o(n)$, where the C-classes are the extension of the B-classes to decision problems in which the inputs can be restricted.

All our results hold not only for the $\mathcal{LOCAL}$ model but also for more restrictive models of computation, such as the $\mathcal{CONGEST}(B)$ model (for $B = O(1)$).

Related Work. The notion of local decision and local verification of languages has received quite a lot of attention recently. In the $\mathcal{LOCAL}$ model, solving a decision problem requires the processors to independently inspect their local neighborhood and collectively decide whether the global instance belongs to some specified language. Inspired by classical computation complexity theory, Fraigniaud et al. [13] suggested that the study of decision problems may lead to new structural insights also in the more complex distributed computing setting. Indeed, following that paper, efforts were made to form a fundamental computational complexity theory for distributed decision problems in various other aspects of distributed computing [13,14,15,16].

The classes LD, NLD and BPLD defined in [13] are the distributed analogues of the classes P, NP and BPP, respectively. The contribution of [13] is threefold: it establishes the impact of nondeterminism, randomization, and randomization + nondeterminism, on local computation. This is done by proving structural results, developing a notion of local reduction and establishing completeness results. One of the main results is the existence of a sharp threshold above which randomization does not help (at least for hereditary languages), and the BPLD classes were classified into two: below and above the randomization threshold.

The current paper "zooms" into the spectrum of classes below the randomization threshold, and defines a infinite hierarchy of BPLD classes, each of which is separated from the class above it in the hierarchy.

The question of whether randomization helps in improving locality for construction problems has been studied extensively. Naor and Stockmeyer [26] considered a subclass of LD($O(1)$), called LCL[3], and studied the question of how to compute in $O(1)$ rounds the constructive versions of decision problems in LCL. The paper demonstrates that randomization does not help, in the sense that if a problem has a local Monte Carlo randomized algorithm, then it also has a local deterministic algorithm. There are several differences between the setting of [26] and ours. First, [26] considers the power of randomization for *constructing* a solution, whereas we study the power of randomization for *deciding* languages[4]. Second, while [26] deals with constant time computations, our separation results apply to arbitrary time computations, potentially depending on the size of the instance (graph and input). The different settings imply different impacts for randomization: while the current paper and [13] show that randomization can indeed help for improving locality of decision problems, [26] shows that randomization does *not* help in constructing a solution for a problem in LCL in constant time. The question of whether randomization helps in local computations was studied for *specific* problems, such as MIS, $(\Delta + 1)$-coloring, and maximal matching [2,5,23,24,25,27,29]. Finally, the classification of decision problems in distributed computing has been studied in several other models. For example, [6] and [18] study specific decision problems in the $\mathcal{CONGEST}$ model. In addition, decision problems have been studied in the asynchrony discipline too, specifically in the framework of *wait-free computation* [15,16] and *mobile agents computing* [14]. In the wait-free model, the main issues are not spatial constraints but timing constraints (asynchronism and faults). The main focus of [16] is deterministic protocols aiming at studying the power of the "decoder", i.e., the interpretation of the results. While this paper essentially considers the AND-checker, (as a global "yes" corresponds to all processes saying "yes"), [16] deals with other interpretations, including more values (not only "yes" and "no"), with the objective of designing checkers that use the smallest number of values.

Preliminaries. In the $\mathcal{LOCAL}$ (respectively $\mathcal{CONGEST}(B)$) model, processors perform in synchronous rounds, where in each round, every processor (1) sends messages of arbitrary (resp., $O(B)$ bits) size to its neighbors, (2) receives messages from its neighbors, and (3) performs arbitrary individual computations. After a number of rounds (that may depend on the network G connecting the processors, and may vary among the processors, since nodes have different

[3] LCL is essentially LD($O(1)$) restricted to languages involving graphs of constant maximum degree and processor inputs taken from a set of constant size.

[4] There is a fundamental difference between such tasks when locality is concerned. Indeed, whereas the validity of constructing a problem in LCL is local (by definition), the validity in our setting is "global", in the sense that in an illegal instance, it is sufficient that at least one vertex in the entire network outputs "no".

identities, potentially different inputs, and are typically located at non-isomorphic positions in the network), every processor v terminates and generates its output.

Consider a distributed (p,q)-decider $\mathcal{A}$ running in a network G with input $\mathbf{x}$ and identity assignment Id (assigning distinct integers to the nodes of G). The output of processor v in this scenario is denoted by $\text{out}_{\mathcal{A}}(G,\mathbf{x},\text{Id},v)$, or simply $\text{out}(v)$ when the parameters are clear from the context. In the case of decision problem, $\text{out}(v) \in \{\text{“yes”}, \text{“no”}\}$ for every processor v.

An n-node path P is represented as a sequence $P = (1,\ldots,n)$, oriented from left to right. (Node i does not know its position in the path.) Given an instance $(P,\mathbf{x})$ with ID's Id and a subpath $S \subset P$, let $\mathbf{x}_S$ (respectively Id_S) be the restriction of $\mathbf{x}$ (resp., Id) to S. We may refer to subpath $S = (i,\ldots,j) \subset P$ as $S = [i,j]$. For a set $U \subseteq V(G)$, let $\mathcal{E}(G,\mathbf{x},\text{Id},U)$ denote the event that, when running $\mathcal{A}$ on $(G,\mathbf{x})$ with id-assignment Id, all nodes in U output “yes”. Given a language $\mathcal{L}$, an instance $(G,\mathbf{x})$ is called *legal* iff $(G,\mathbf{x}) \in \mathcal{L}$. Given a time bound t, a subpath $S = [i,j]$ is called an *internal* subpath of P if $i \geq t+2$ and $j \leq n-t-1$. Note that if the subpath S is internal to P, then when running a t-round algorithm, none of the nodes in S “sees” the endpoints of P.

The following concept is crucial to the proofs of our separation results.

Definition 1. *Let S be a subpath of P. For $\delta \in [0,1]$, S is said to be a (δ,λ)-*secure *subpath if $|S| \geq \lambda$ and $\Pr[\mathcal{E}(P,\mathbf{x},\text{Id},V(S))] \geq 1-\delta$.*

We typically use (δ,λ)-secure subpaths for values of $\lambda \geq 2t+1$ where t is the running time of the (p,q)-decider $\mathcal{A}$ on $(P,\mathbf{x})$ for some fixed identity assignment Id. Indeed, it is known [13] that if $(P,\mathbf{x}) \in \mathcal{L}$, then every long enough subpath S of P contains an internal (δ,λ)-secure subpath S'. More precisely, define

$$\ell(\delta,\lambda) = 4(\lambda+2t)\lceil \log p/\log(1-\delta)\rceil. \tag{1}$$

Fact 1 ([13]). *Let $(P,\mathbf{x}) \in \mathcal{L}$, $\delta \in [0,1]$, $\lambda \geq 1$. Then for every $\ell(\delta,\lambda)$-length subpath S there is a subpath S' (internal to S) that is (δ,λ)-secure.*

To avoid cumbersome notation, when $\lambda = 2t+1$, we may omit it and refer to $(\delta,2t+1)$-secure subpaths as δ-secure subpaths. In addition, set

$$\ell(\delta) := \ell(\delta,2t+1).$$

Let us next illustrate a typical use of Fact 1. Recall that t denotes the running time of the (p,q)-decider $\mathcal{A}$ on $(P,\mathbf{x}) \in \mathcal{L}$ with IDs Id. Let S be a subpath of P of length $\ell(\delta)$. Denote by L (resp., R) the subpath of P to the “left” (resp., “right”) of S. Informally, if the length of S is larger than $2t+1$, then S serves as a separator between the two subpaths L and R. This follows since as algorithm $\mathcal{A}$ runs in t rounds, each node in P is affected only by its t neighborhood. As the t neighborhood of every node $u \in L$ and $v \in R$ do not intersect, the events $\mathcal{E}(P,\mathbf{x},\text{Id},L)$ and $\mathcal{E}(P,\mathbf{x},\text{Id},R)$ are independent.

The security property becomes useful when upper bounding the probability that at least some node in P says “no”, by applying a union bound on the events

$\mathcal{E}(P, \mathbf{x}, \mathrm{Id}, V(L) \cup V(R))$ and $\mathcal{E}(P, \mathbf{x}, \mathrm{Id}, V(S))$. Denoting the event complementary to $\mathcal{E}(P, \mathbf{x}, \mathrm{Id}, V(P))$, by $\mathcal{E}'$ we have

$$\begin{aligned}\Pr[\mathcal{E}'] &= 1 - \Pr[\mathcal{E}(P, \mathbf{x}, \mathrm{Id}, V(P))] \\ &\leq (1 - \Pr[\mathcal{E}(P, \mathbf{x}, \mathrm{Id}, V(L))] \cdot \Pr[\mathcal{E}(P, \mathbf{x}, \mathrm{Id}, V(R))]) \\ &\quad + (1 - \Pr[\mathcal{E}(P, \mathbf{x}, \mathrm{Id}, V(S))]) \\ &\leq 1 - \Pr[\mathcal{E}(P, \mathbf{x}, \mathrm{Id}, V(L))] \cdot \Pr[\mathcal{E}(P, \mathbf{x}, \mathrm{Id}, V(R))] + \delta\ .\end{aligned}$$

The specific choice of λ and δ depends on the context. Informally, the guiding principle is to set δ small enough so that the role of the central section S can be neglected, while dealing separately with the two extreme sections L and R become manageable for they are sufficiently far apart.

2 The B_k Hierarchy Is Strict

In this section we show that the classes B_k, $k \geq 1$, form an infinite hierarchy of distinct classes, thereby proving that the general ability to boost the probability of success for a randomized decision problem is quite limited. In fact, we show separation in a very strong sense: there are decision problems in $B_{k+1}(0)$, i.e., that have a (p, q)-decider running in zero rounds with $p^{1+1/(k+1)} + q > 1$, which cannot be decided by a (p, q)-decider with $p^{1+1/k} + q > 1$, even if the number of rounds of the latter is as large as $n^{1-\varepsilon}$ for every fixed $\varepsilon > 0$.

Theorem 2. *$B_{k+1}(0) \setminus B_k(t) \neq \emptyset$ for every $k \geq 1$ and every $t = o(n)$.*

Proof. Let k be any positive integer. We consider the following distributed language, which is a generalized variant `AMOS-`k of the problem `AMOS` introduced in [13]. As in `AMOS`, the input $\mathbf{x}$ of `AMOS-`k satisfies $\mathbf{x} \in \{0, 1\}^n$, i.e., each node v is given as input a boolean $\mathbf{x}(v)$. The language `AMOS-`k is then defined by:

$$\texttt{At-Most-k-Selected}\ (\texttt{AMOS-}k) = \{(G, \mathbf{x}) \text{ s.t. } \parallel \mathbf{x} \parallel_1 \leq k\}.$$

Namely, `AMOS-`k consists of all instances containing at most k selected nodes (i.e., at most k nodes with input 1), with all other nodes unselected (having input 0). In order to prove Theorem 2, we show that `AMOS-`$k \in B_{k+1}(0) \setminus B_k(t)$ for every $t = o(n)$.

We first establish that `AMOS-`k belongs to $B_{k+1}(0)$. We adapt algorithm $\mathcal{A}$ presented in [13] for `AMOS` to the case of `AMOS-`k. The following simple randomized algorithm runs in 0 time: every node v which is not selected, i.e., such that $\mathbf{x}(v) = 0$, says "yes" with probability 1; and every node which is selected, i.e., such that $\mathbf{x}(v) = 1$, says "yes" with probability $p^{1/k}$, and "no" with probability $1 - p^{1/k}$. If the graph has $s \leq k$ nodes selected, then all nodes say "yes" with probability $p^{s/k} \geq p$, as desired. On the other hand, if there are $s \geq k+1$ selected nodes, then at least one node says "no" with probability $1 - p^{s/k} \geq 1 - p^{(k+1)/k} = 1 - p^{1+1/k}$. We therefore get a (p, q)-decider with $p^{1+1/k} + q \geq 1$, that is, such that $p^{1+1/(k+1)} + q > 1$. Thus `AMOS-`$k \in B_{k+1}(0)$.

We now consider the harder direction, and prove that $\texttt{AMOS-}k \notin B_k(t)$, for any $t = o(n)$. To prove this separation, it is sufficient to consider $\texttt{AMOS-}k$ restricted to the family of n-node paths. Fix a function $t = o(n)$, and assume, towards contradiction, that there exists a distributed (p, q)-decider $\mathcal{A}$ for $\texttt{AMOS-}k$ that runs in $O(t)$ rounds, with $p^{1+1/k} + q > 1$. Let $\varepsilon \in (0, 1)$ be such that $p^{1+1/k+\varepsilon} + q > 1$. Let P be an n-node path, and let $S \subset P$ be a subpath of P. Let $\delta \in [0, 1]$ be a constant satisfying

$$0 < \delta < p^{1+1/k}\,(1 - p^{\varepsilon})\,/k \ . \tag{2}$$

Consider a positive instance and a negative instance of $\texttt{AMOS-}k$, respectively denoted by $I = (P, \mathbf{x})$ and $I' = (P, \mathbf{x}')$. Both instances are defined on the same n-node path P, where $n \geq k\,(\ell(\delta) + 1) + 1$. Recall that $\ell(\delta) = \ell(\delta, 2t + 1)$ (see Eq. (1)). We consider executions of $\mathcal{A}$ on these two instances, where nodes are given the same id's. Both instances have almost the same input. In particular, the only difference is that instance I contains k selected nodes, whereas I' has the same selected nodes as I plus one additional selected node. Therefore I is legal, while I' is illegal. In I', the path P is composed of $k + 1$ sections, each containing a unique selected node, and where each pair of consecutive sections separated by a δ-secure subpath. More precisely, let us enumerate the nodes of P from 1 to n, with node v adjacent to nodes $v - 1$ and $v + 1$, for every $1 < v < n$. Consider the k subpaths of P defined by: $S_i = [(i - 1)\ell(\delta) + i + 1, i \cdot \ell(\delta) + i]$ for $i = \{1, \ldots, k\}$. Let the selected nodes in I' be positioned as follows. Let $u_1 = 1$ and let $u_i = (i - 1)\ell(\delta) + i$ for $i = 2, \ldots, k + 1$. Then set

$$\mathbf{x}'(v) = \begin{cases} 1, \text{ if } v = u_i \text{ for some } i \in \{1, ..., k + 1\} \\ 0, \text{ otherwise.} \end{cases}$$

See Fig. 1(a) for a schematic representation of I'.

Our next goal is to define the legal instance $I = (P, \mathbf{x})$. To do so, we begin by claiming that each S_i contains a δ-secure internal subpath $S'_i = [a_i, b_i]$. Naturally, we would like to employ Fact 1. However, Fact 1 refers to subpaths of *valid* instances $(P, \mathbf{x}) \in \mathcal{L}$, and I' is illegal. So instead, let us focus on the instance $(S_i, \mathbf{x}'_{S_i})$. Since $(S_i, \mathbf{x}'_{S_i})$ contains no leaders, $\parallel \mathbf{x}'_{S_i} \parallel_1 = 0$, it follows that $(S_i, \mathbf{x}'_{S_i}) \in \mathcal{L}$, and Fact 1 can be applied on it. Subsequently, since $|S_i| > \ell(\delta)$ it follows that S_i contains an internal δ-secure subpath $S'_i = [a_i, b_i]$, whose t neighborhood is strictly in S_i. Therefore, when applying algorithm $\mathcal{A}$ on $(S_i, \mathbf{x}'_{S_i}, \mathrm{Id}_{S_i})$ and on $(P, \mathbf{x}', \mathrm{Id})$, the nodes in the $(2t + 1)$-length segment S'_i behave the same, thus $\Pr[\mathcal{E}(P, \mathbf{x}', \mathrm{Id}, V(S'_i))] = \Pr[\mathcal{E}(S_i, \mathbf{x}'_{S_i}, \mathrm{Id}_{S_i}, V(S'_i))]$. Hence, S'_i is a δ-secure subpath in $(P, \mathbf{x}', \mathrm{Id})$ as well, for every $i \in \{1, ..., k\}$, see Fig. 1(b).

The δ-secure subpaths S'_i's are now used to divide P into $2k + 1$ segments. Specifically, there are $k + 1$ segments T_i, $i = 1, \ldots, k + 1$, each with one selected node. The δ-secure subpaths $S'_i = [a_i, b_i]$ separate T_i from T_{i+1}. More precisely, set $T_1 = [1, a_1 - 1]$, $T_i = [b_{i-1} + 1, a_i - 1]$ for $i \in 2, ..., k$, and $T_{k+1} = [b_k + 1, n]$, getting $P = T_1 \circ S'_1 \circ T_2 \circ S'_2 \circ \ldots \circ T_k \circ S'_k \circ T_{k+1}$ where $\circ$ denotes path concatenation. Let $\mathcal{T}_i = \mathcal{E}(P, \mathbf{x}', \mathrm{Id}, V(T_i))$ be the event that all nodes in the subpath T_i say "yes" in the instance I', for $i \in \{1, ..., k+1\}$ and let $p_i = \Pr[\mathcal{T}_i]$ be its probability. Let j be such that $p_j = \max_i p_i$. We now define the valid instance $I = (P, \mathbf{x})$:

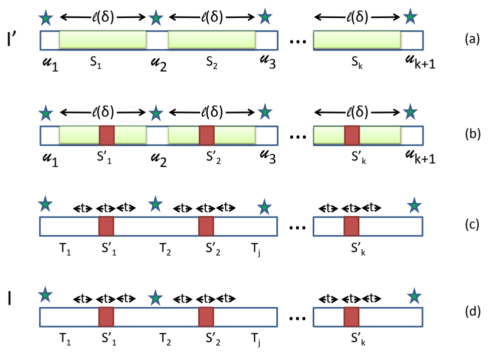


Fig. 1. Illustration of the constructions for Theorem 2. (a) The instance $I' = (P, \mathbf{x}')$ with $k+1$ leaders separated by $\ell(\delta)$-length segements, S_i. (b) The δ-secure subpaths S_i' in each S_i are internal to S_i. (c) The leader-segments T_i interleaving with δ-secure subpaths S_i'. (d) The legal instance $I = (P, \mathbf{x})$, the j^{th} leader of I' is discarded, resulting in a k leader instance.

$$\mathbf{x}(v) = \begin{cases} 1, \text{ if } v = u_i \text{ for some } i \in \{1, ..., k+1\},\ i \neq j, \\ 0, \text{ otherwise.} \end{cases}$$

Note that $\| \mathbf{x}' \|_1 = k+1$ and $\| \mathbf{x} \|_1 = k$, thus $I \in \texttt{AMOS-}k$ while $I' \notin \texttt{AMOS-}k$. See Fig. 1(c,d) for an illustration of I versus I'.

We now make the following observation (the proof defer to full version).

Claim. $\forall i \neq j,\ \Pr[\mathcal{E}(P, \mathbf{x}, \mathrm{Id}, V(T_i))] = p_i$.

Let $\mathcal{N}$ (resp., $\mathcal{N}'$) be the event that there exists at least one node in I (resp., I') that says "no" when applying algorithm $\mathcal{A}$. Similarly, let $\mathcal{Y}$ (resp., $\mathcal{Y}'$) be the event stating that all nodes in the configuration I (resp., I') say "yes". Let $\mathcal{T} = \bigcup_{i=1}^{k+1} \mathcal{T}_i$ be the event that all nodes in each subpaths T_i, for $i \in \{1, ..., k+1\}$ say "yes" in the instance I'. For every $i \in \{1, ..., k\}$, let $\mathcal{S}_i = \mathcal{E}(P, \mathbf{x}', \mathrm{Id}, V(S_i'))$ be the event that all nodes in the δ-secure subpath S_i' say "yes" in the instance I'. We have $\Pr(\mathcal{Y}) = \Pr[\mathcal{E}(P, \mathbf{x}, \mathrm{Id}, V(P))]$ and $\Pr(\mathcal{Y}') = \Pr[\mathcal{E}(P, \mathbf{x}', \mathrm{Id}, V(P))$, while $\Pr(\mathcal{N}) = 1 - \Pr(\mathcal{Y})$ and $\Pr(\mathcal{N}') = 1 - \Pr(\mathcal{Y}')$.

Since $\mathcal{A}$ is a (p, q)-decider, as we assume by contradiction that $\texttt{AMOS-}k$ in B_k, we have $\Pr(\mathcal{N}') \geq q$, and thus $\Pr(\mathcal{N}') > 1 - p^{1+1/k+\varepsilon}$. Therefore, $\Pr(\mathcal{Y}') < p^{1+1/k+\varepsilon}$. Moreover, since $I \in \texttt{AMOS-}k$, we also have that $\Pr(\mathcal{Y}) \geq p$. Therefore, the ratio $\hat{\rho} = \Pr(\mathcal{Y}')/\Pr(\mathcal{Y})$ satisfies

$$\hat{\rho} < p^{1/k+\varepsilon} \ . \tag{3}$$

On the other hand, note that by applying the union bound to the $k+1$ events $\mathcal{T}, \bigcup_{i=1}^{k} \mathcal{S}_i$, we get

$$\Pr(\mathcal{N}') \leq (1 - \Pr[\mathcal{T}]) + \left(\sum_{i=1}^{k} (1 - \Pr[\mathcal{S}_i]) \right) \leq 1 - p_j \cdot \prod_{i \neq j} p_i + k \cdot \delta,$$

where the last inequality follows by the fact that each S'_i is a $(\delta, 2t+1)$-secure subpath, thus the events $\mathcal{T}_{i_1}, \mathcal{T}_{i_2}$ are independent for every $i_1, i_2 \in \{1, ..., k+1\}$ (since the distance between any two nodes $u \in T_{i_1}$ and $v \in T_{i_2}$ is at least $2t+1$). This implies that $\Pr(\mathcal{Y}') \geq p_j \cdot \prod_{i \neq j} p_i - k \cdot \delta$. Since $\Pr(\mathcal{Y}) \leq \prod_{i \neq j} p_i$ (by the independence of the events $\mathcal{T}_{i_1}, \mathcal{T}_{i_2}$, for every $i_1, i_2 \in \{1, ...k+1\}$), it then follows that the ratio $\hat{\rho}$ satisfies

$$\hat{\rho} \geq \frac{p_j \cdot \prod_{i \neq j} p_i - k \cdot \delta}{\prod_{i \neq j} p_i} \geq p_j - \frac{k \cdot \delta}{\prod_{i \neq j} p_i} \geq p_j - k \cdot \delta / p \ , \tag{4}$$

where the last inequality follows by the fact that $I \in \texttt{AMOS-}k$ and thus $\prod_{i \neq j} p_i \geq \Pr(\mathcal{Y}) \geq p$. Finally, note that $p_j \geq p^{1/k}$. This follows since $p_j \geq p_i$ for every $i \in \{1, ..., k+1\}$, so $p_j^k \geq \prod_{i \neq j} p_i \geq p$. By Eq. (4), we then have that $\hat{\rho} \geq p^{1/k} - k \cdot \delta / p$. Combining this with Eq. (3), we get that $p^{1/k} - k \cdot \delta / p < p^{1/k+\varepsilon}$, which is in contradiction to the definition of δ in Eq. (2).

Finally, we show that the $B_k(t)$ hierarchy does not capture all languages even for $k = \infty$ and t as large as $o(n)$. Due to space limitations, the proof of the following theorem is deferred to the full version of this paper.

Theorem 3. *There is a language not in $B_\infty(t)$, for every $t = o(n)$.*

3 A Sharp Determinism - Randomization Threshold

It is known [13] that beyond the threshold $p^2 + q = 1$, randomization does not help. This result however holds only for a particular type of languages, called *hereditary*, i.e., closed under inclusion. In this section, we provide one more evidence supporting our belief that the threshold $p^2 + q = 1$ identified in [13] holds for *all* languages, and not only for hereditary languages. Indeed, we prove that, restricted to path topologies and finite inputs, *every* language $\mathcal{L}$ for which there exists a (p, q)-decider running in constant time, with $p^2 + q > 1$, can actually be decided deterministically in constant time.

Theorem 4. *Let $\mathcal{L}$ be a distributed language restricted to paths, with a finite set of input values. If $\mathcal{L} \in B_1(O(1))$, then $\mathcal{L} \in \mathrm{LD}(O(1))$.*

Proof. Let $\mathcal{L} \in B_1(O(1))$ be a distributed language restricted to paths, and defined on the (finite) input set Σ. Consider a distributed (p, q)-decider $\mathcal{A}$ for $\mathcal{L}$ that runs in $t = O(1)$ rounds, with $p^2 + q > 1$. Fix a constant δ such that $0 < \delta < p^2 + q - 1$.

Given a subpath S of a path P, let us denote by S_l (respectively, S_r) the subpath of P to the left (resp., right) of S, so that $P = S_l \circ S \circ S_r$.

Informally, a collection of three paths P, P', and P'' (of possibly different lengths) is called a *λ-path triplet* if (1) the inputs of those paths agree on some "middle" subpath of size at least λ, (2) paths P and P'' coincide on their corresponding "left" parts, and (3) paths P' and P'' coincide on their "right" parts.

See Fig. 2. Formally, a λ-*path triplet* is a triplet $[(P,S,\mathbf{x}),(P',S',\mathbf{x}'),(P'',S'',\mathbf{x}'')]$ such that $|P|,|P'|,|P''|\geq\lambda$, $\mathbf{x},\mathbf{x}',\mathbf{x}''$ are inputs on these paths, respectively, and $S\subset P$, $S'\subset P'$, $S''\subset P''$ are three subpaths satisfying (1) $|S|=|S'|=|S''|\geq\lambda$, (2) $\mathbf{x}_S=\mathbf{x}'_{S'}=\mathbf{x}''_{S''}$, and (3) $\mathbf{x}''_{S''_l}=\mathbf{x}_{S_l}$ and $\mathbf{x}''_{S''_r}=\mathbf{x}'_{S'_r}$.

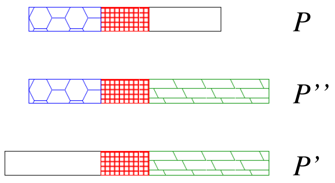


Fig. 2. Example of a λ-path triplet (the red zone is of length at least λ)

Claim. Let $[(P,S,\mathbf{x}),(P',S',\mathbf{x}'),(P'',S'',\mathbf{x}'')]$ be a λ-path triplet. If $\lambda\geq\ell(\delta)$, for ℓ as defined in Eq. (1), then $\Big((P,\mathbf{x})\in\mathcal{L}$ and $(P',\mathbf{x}')\in\mathcal{L}\Big)\Rightarrow(P'',\mathbf{x}'')\in\mathcal{L}$.

Proof. Consider an identity assignment Id'' for $(P'',\mathbf{x}'')$. Let Id and Id' be identity assignments for $(P,\mathbf{x})$, and $(P',\mathbf{x}')$, respectively, which agree with Id'' on the corresponding nodes. That is: (a) assignments Id, Id', and Id'' agree on the nodes in S, S' and S'', respectively; (b) Id and Id'' agree on the nodes in S_ℓ and S''_ℓ, respectively; and (c) Id and Id'' agree on the nodes in S'_r and S''_r, respectively. Since $(P,\mathbf{x})\in\mathcal{L}$, and since $|S|=\lambda\geq\ell(\delta)$, it follows from Fact 1 that S contains an internal δ-secure subpath H. Then, let H' and H'' be the subpaths of P' and P'' corresponding to H. Since S and S'' coincide in their inputs and identity assignments, then H,H',H'' have the same t-neighborhood in P,P',P'' respectively. Hence, H'' is also a δ-secure (when running algorithm $\mathcal{A}$ in instance $(P'',\mathbf{x}'')$). Since both $(P,\mathbf{x})$ and $(P',\mathbf{x}')$ belong to $\mathcal{L}$, we have

$$\Pr[\mathcal{E}(H''_\ell,\mathrm{Id}'',\mathbf{x}'')=\Pr[\mathcal{E}(H_\ell,\mathrm{Id},\mathbf{x})])]\geq p$$
$$\Pr[\mathcal{E}(H''_r,\mathrm{Id}'',\mathbf{x}'')=\Pr[\mathcal{E}(H'_r,\mathrm{Id}',\mathbf{x}')])]\geq p.$$

Moreover, as $|H''|\geq 2t+1$, the two events $\mathcal{E}(H''_\ell,\mathrm{Id}'',\mathbf{x}'')$ and $\mathcal{E}(H''_r,\mathrm{Id}'',\mathbf{x}'')$ are independent. Hence $\Pr[\mathcal{E}(H''_\ell\cup H''_r,\mathrm{Id}'',\mathbf{x}'')]\geq p^2$. In other words, the probability that some node in $H''_\ell\cup H''_r$ says "no" is at most $1-p^2$. It follows, by union bound, that the probability that some node in H'' says "no" is at most $1-p^2+\delta<q$. Since $\mathcal{A}$ is a (p,q)-decider for $\mathcal{L}$, it cannot be the case that $(H'',\mathbf{x}'')\notin\mathcal{L}$.

We now observe that, w.l.o.g., one can assume that in all instances $(P,\mathbf{x})$ of $\mathcal{L}$, the two extreme vertices of the path P have a special input symbol $\otimes$. To see why this holds, let $\otimes$ be a symbol not in Σ, and consider the following language $\mathcal{L}'$ defined over $\Sigma\cup\{\otimes\}$. Language $\mathcal{L}'$ consists of instances $(P,\mathbf{x})$ such that (1) the endpoints of P have input $\otimes$, and (2) $(P',\mathbf{x}')\in\mathcal{L}$, where P' is the path resulting from removing the endpoints of P, and where $\mathbf{x}'_v=\mathbf{x}_v$ for every node v of P'. Any (p,q) decider algorithm for $\mathcal{L}$ (resp., $\mathcal{L}'$), can be trivially transformed into a (p,q) decider algorithm for $\mathcal{L}'$ (resp., $\mathcal{L}$) with the same success guarantees and running time. Hence, in the remaining of the proof, we assume that in all instances $(P, \mathbf{x}) \in \mathcal{L}$, the two extreme vertices of the path P have input $\otimes$.

A given instance $(P, \mathbf{x})$ is *extendable* if there exists an extension of it in $\mathcal{L}$, i.e., if there exists an instance $(P', \mathbf{x}') \in \mathcal{L}$ such that $P \subseteq P'$ and $\mathbf{x}'_P = \mathbf{x}$.

Claim. There exists a (centralized) algorithm $\mathcal{X}$ that, given any configuration $(P, \mathbf{x})$ with $|P| \leq 2\ell(\delta) + 1$, decides whether $(P, \mathbf{x})$ is extendable. Moreover, algorithm $\mathcal{X}$ uses messages of constant size.

We may assume, hereafter, that such an algorithm $\mathcal{X}$, as promised by Claim 3, is part of the language specification given to the nodes, each node then can verify by a *local* computation if the instance restricted to its $\ell(\delta)$ neighborhood is extendable. We show that $\mathcal{L} \in \text{LD}(O(t))$ by proving the existence of a deterministic algorithm $\mathcal{D}$ that recognizes $\mathcal{L}$ in $O(t)$ rounds. Given a path P, an input $\mathbf{x}$ over P, and an identity assignment Id, algorithm $\mathcal{D}$ applied at a node u of P operates as follows. If $\mathbf{x}_u = \otimes$ then u outputs "yes" if and only if u is an endpoint of P. Otherwise, i.e., if $\mathbf{x}_u \neq \{\otimes\}$, then u outputs "yes" if and only if $(\mathtt{b}_u, \mathbf{x}_{\mathtt{b}_u})$ is extendable (using algorithm $\mathcal{X}$), where $\mathtt{b}_u = \mathtt{b}(u, \ell(\delta))$ is the ball centered at u, and of radius $\ell(\delta)$ in P.

Algorithm $\mathcal{D}$ is a deterministic algorithm that runs in $\ell(\delta)$ rounds. We claim that Algorithm $\mathcal{D}$ recognizes $\mathcal{L}$. To establish that claim, consider first an instance $(P, \mathbf{x}) \in \mathcal{L}$. For every node u, $(P, \mathbf{x}) \in \mathcal{L}$ is an extension of $(\mathtt{b}_u, \mathbf{x}_{\mathtt{b}_u})$. Therefore, every node u outputs "yes", as desired. Now consider an instance $(P, \mathbf{x}) \notin \mathcal{L}$. Assume, for the purpose of contradiction, that there exists an identity assignment Id such that, when applying $\mathcal{D}$ on $(P, \mathbf{x}, \text{Id})$, every node u outputs "yes".

Claim. In this case, $|P| > 2\ell(\delta) + 1$.

Let $S \subseteq P$ be the longest subpath of P such that there exists an extension $(P', \mathbf{x}')$ of $(S, \mathbf{x}_S)$, with $(P', \mathbf{x}') \in \mathcal{L}$. Since $|P| > 2\ell(\delta) + 1$, and since the middle node of P outputs "yes", we have $|S| \geq 2\ell(\delta) + 1$. The proof carries on by distinguishing two cases for the length of S. If $S = P$, then $(P, \mathbf{x})$ can be extended to $(P', \mathbf{x}') \in \mathcal{L}$. By the same arguments as above, since each extremity w of P has input $\otimes$, we conclude that $P = P'$, with $\mathbf{x} = \mathbf{x}'$. Contradicting the fact that $(P, \mathbf{x}) \notin \mathcal{L}$. Therefore $2\ell(\delta) + 1 \leq |S| < |P|$. Let a and b be such that $S = [a, b]$. As S is shorter than P, it is impossible for both a and b to be endpoints of P. Without loss of generality, assume that a is not an endpoint of P. Since a outputs "yes", there exists an extension $(P'', \mathbf{x}'') \in \mathcal{L}$ of $(\mathtt{b}_a, \mathbf{x}_{\mathtt{b}_a})$. In fact, $(P'', \mathbf{x}'')$ is also an extension of $\mathbf{x}_{[a, a+\ell(\delta)]}$. Since $\mathbf{x}'$ and $\mathbf{x}''$ agree on $[a, a + \ell(\delta)]$, and since both $(P', \mathbf{x}')$, and $(P'', \mathbf{x}'')$ are in $\mathcal{L}$, we get from Lemma 3 that $\mathbf{x}_{[a-1,b]}$ can be extended to an input $(P''', \mathbf{x}''') \in \mathcal{L}$, which contradicts the choice of S. The theorem follows.

4 On the Impossibility of Boosting

Theorems 2 and 3 demonstrate that boosting the probability of success might be doable, but only from (p, q) satisfying $p^{1+1/(k+1)} + q > 1$ to (p, q) satisfying

$p^{1+1/k} + q > 1$ (with the extremes excluded). In this section, we prove that once the inputs may be restricted in certain ways, the ability to boost the success probability become almost null. More precisely, recall that so far we considered languages as collections of pairs $(G, \mathbf{x})$ where G is a (connected) n-node graph and $\mathbf{x} \in \Sigma^n$ is the input vector to the nodes of G, in some finite of infinite alphabet Σ, that is, $\mathbf{x}(v) \in \Sigma$ for all $v \in V(G)$. An instance of an algorithm $\mathcal{A}$ deciding a language $\mathcal{L}$ was defined as *any* such pair $(G, \mathbf{x})$. We now consider the case where the set of instances is restricted to some specific subset of inputs $\mathcal{I} \subset \Sigma^n$. That is, the distributed algorithm $\mathcal{A}$ has now the *promise* that in the instances $(G, \mathbf{x})$ admissible as inputs, the input vector $\mathbf{x}$ is restricted to $\mathbf{x} \in \mathcal{I} \subset \Sigma^n$.

We define the classes $C_r(t)$ in a way identical to the classes $B_k(t)$, but generalized in two ways. First, the parameter r is not bounded to be integral, but can be any positive real. Second, the decision problems under consideration are extended to the ones in which the set of input vectors $\mathbf{x}$ can be restricted. So, in particular, $B_k(t) \subseteq C_k(t)$, for every positive integer k, and every function t. The following theorem proves that boosting can made as limited as desired.

Theorem 5. *Let $r < r'$ be any two positive reals. Then, $C_{r'}(0) \setminus C_r(t) \neq \emptyset$ for every $t = o(n)$.*

Note that Theorem 5 demonstrates not only the (almost) inability of boosting the probability of success when the inputs to the nodes are restricted to specific kinds, but also the inability of derandomizing, even above the threshold $p^2 + q = 1$. Indeed, the following is a direct consequence of Theorem 5.

Corollary 1. *For every positive real r, there is a decision problem in $C_r(0)$ which cannot be decided* deterministically *in $o(n)$ rounds.*

References

1. Afek, Y., Kutten, S., Yung, M.: The local detection paradigm and its applications to self stabilization. Theoretical Computer Science 186(1-2), 199–230 (1997)
2. Alon, N., Babai, L., Itai, A.: A fast and simple randomized parallel algorithm for the maximal independent set problem. J. Algorithms 7(4), 567–583 (1986)
3. Amit, A., Linial, N., Matousek, J., Rozenman, E.: Random lifts of graphs. In: Proc. 12th ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 883–894 (2001)
4. Awerbuch, B., Patt-Shamir, B., Varghese, G.: Self-Stabilization By Local Checking and Correction. In: Proc. IEEE Symp. on the Foundations of Computer Science (FOCS), pp. 268–277 (1991)
5. Barenboim, L., Elkin, M.: Distributed $(\Delta + 1)$-coloring in linear (in delta) time. In: Proc. 41st ACM Symp. on Theory of computing (STOC), pp. 111–120 (2009)
6. 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. 43rd ACM Symp. on Theory of Computing, STOC (2011)
7. Dereniowski, D., Pelc, A.: Drawing maps with advice. Journal of Parallel and Distributed Computing 72, 132–143 (2012)
8. Dijkstra, E.W.: Self-stabilization in spite of distributed control. Comm. ACM 17(11), 643–644 (1974)

9. Dolev, S., Gouda, M., Schneider, M.: Requirements for silent stabilization. Acta Informatica 36(6), 447–462 (1999)
10. Fraigniaud, P., Gavoille, C., Ilcinkas, D., Pelc, A.: Distributed Computing with Advice: Information Sensitivity of Graph Coloring. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 231–242. Springer, Heidelberg (2007)
11. Fraigniaud, P.: D Ilcinkas, and A. Pelc. Communication algorithms with advice. J. Comput. Syst. Sci. 76(3-4), 222–232 (2008)
12. Fraigniaud, P.: A Korman, and E. Lebhar. Local MST computation with short advice. In: Proc. 19th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA), pp. 154–160 (2007)
13. Fraigniaud, P., Korman, A., Peleg, D.: Local Distributed Decision. In: Proc. 52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 708–717 (2011)
14. 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)
15. Fraigniaud, P., Rajsbaum, S., Travers, C.: Locality and Checkability in Wait-Free Computing. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 333–347. Springer, Heidelberg (2011)
16. Fraigniaud, P., Rajsbaum, S., Travers, C.: Universal Distributed Checkers and Orientation-Detection Tasks (submitted, 2012)
17. Göös, M., Suomela, J.: Locally checkable proofs. In: Proc. 30th ACM Symp. on Principles of Distributed Computing, PODC (2011)
18. Kor, L., Korman, A., Peleg, D.: Tight Bounds For Distributed MST Verification. In: Proc. 28th Int. Symp. on Theoretical Aspects of Computer Science, STACS (2011)
19. Korman, A., Kutten, S.: Distributed verification of minimum spanning trees. Distributed Computing 20, 253–266 (2007)
20. Korman, A., Kutten, S., Masuzawa, T.: Fast and Compact Self-Stabilizing Verification, Computation, and Fault Detection of an MST. In: Proc. 30th ACM Symp. on Principles of Distributed Computing, PODC (2011)
21. Korman, A., Kutten, S.: D Peleg. Proof labeling schemes. Distributed Computing 22, 215–233 (2010)
22. Korman, A., Sereni, J.S., Viennot, L.: Toward More Localized Local Algorithms: Removing Assumptions Concerning Global Knowledge. In: Proc. 30th ACM Symp. on Principles of Distributed Computing (PODC), pp. 49–58 (2011)
23. Kuhn, F.: Weak graph colorings: distributed algorithms and applications. In: Proc. 21st ACM Symp. on Parallel Algorithms and Architectures (SPAA), pp. 138–144 (2009)
24. Luby, M.: A simple parallel algorithm for the maximal independent set problem. SIAM J. Comput. 15, 1036–1053 (1986)
25. Naor, M.: A Lower Bound on Probabilistic Algorithms for Distributive Ring Coloring. SIAM J. Discrete Math. 4(3), 409–412 (1991)
26. Naor, M., Stockmeyer, L.: What can be computed locally? SIAM J. Comput. 24(6), 1259–1277 (1995)
27. Panconesi, A., Srinivasan, A.: On the Complexity of Distributed Network Decomposition. J. Algorithms 20(2), 356–374 (1996)
28. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. SIAM (2000)
29. Schneider, J., Wattenhofer, R.: A new technique for distributed symmetry breaking. In: Proc. 29th ACM Symp. on Principles of Distributed Computing (PODC), pp. 257–266 (2010)

The Strong At-Most-Once Problem

Sotirios Kentros[1,⋆], Chadi Kari[2], and Aggelos Kiayias[1,⋆⋆]

[1] Computer Science and Engineering, University of Connecticut, Storrs, CT, USA
skentros@engr.uconn.edu, aggelos@kiayias.com

[2] Computer Science, Bridgewater State University, Bridgewater, MA, USA
celkari@bridgew.edu

Abstract. The at-most-once problem in shared memory asks for the completion of a number of tasks by a set of independent processors while adhering to "at most once" semantics. At-most-once algorithms are evaluated in terms of *effectiveness*, which is a measure that expresses the total number of tasks completed at-most-once in the worst case. Motivated by the lack of deterministic solutions with high effectiveness, we study the feasibility of (a close variant of) this problem. The *strong at most once* problem is solved by an at-most-one algorithm when all tasks are performed if no participating processes crash during the execution of the algorithm. We prove that the strong at-most-once problem has consensus number 2. This explains, via impossibility, the lack of wait-free deterministic solutions with high effectiveness for the at most once problem using only read/write atomic registers. We then present the first k-adaptive effectiveness optimal randomized solution for the strong at-most-once problem, that has optimal expected work for a non-trivial number of participating processes. Our solution also provides the first k-adaptive randomized solution for the Write-All problem, a dual problem to at-most-once.

1 Introduction

The *at-most-once problem* for asynchronous shared memory systems was introduced by Kentros *et al.* [18] as the problem of performing a set of n jobs by m fail-prone processes while maintaining at-most-once semantics.

The *at-most-once* semantic for object invocation ensures that an operation accessing and altering the state of an object is performed no more than once. This semantic is among the standard semantics for remote procedure calls (RPC) and method invocations and it provides important means for reasoning about the safety of critical applications. Uniprocessor systems may trivially provide solutions for at-most-once semantics by implementing a central schedule for operations. The problem becomes very challenging for autonomous processes in a shared-memory system with concurrent invocations on multiple objects. At-most-once semantics have been thoroughly studied in the context of at-most-once

⋆ Research supported in part by the State Scholarships Foundation of Greece.

⋆⋆ Research supported in part by NSF awards 0831304, 0831306 and and EU projects RECUP and CODAMODA.

message delivery [7,21,23] and at-most-once process invocation for RPC [6,22,28]. However, finding effective solutions for asynchronous shared-memory multiprocessors, in terms of how many at-most-once invocations can be performed by the cooperating processes, is largely an open problem. Solutions for the at-most-once problem, using only atomic read/write memory, and without specialized hardware support such as conditional writing, provide a useful tool in reasoning about the safety properties of applications developed for a variety of multiprocessor systems, including those not supporting bus-interlocking instructions and multi-core systems. Specifically, in recent years, attention has shifted from increasing clock speed towards *chip multiprocessing*, in order to increase the performance of systems. Because of the differences in each multi-core system, asynchronous shared memory is becoming an important abstraction for arguing about the safety properties of parallel applications in such systems. In the next years, we expect chip multiprocessing to appear in a wide range of applications, many of which will have components that need to satisfy at-most-once semantics in order to guarantee safety. Such applications may include autonomous robotic devices, robotic devices for assisted living, automation in production lines or medical facilities. In such applications performing specific tasks at-most-once may be of paramount importance for safety of patients, the workers in a facility, or the devices themselves. Such tasks could be the triggering of a motor in a robotic arm, the activation of the X-ray gun in an X-ray machine, or supplying a dosage of medicine to a patient.

The definition of the at-most-once problem by Kentros *et al.* [18] is general and allows a variety of solutions, including trivial ones where all, or some processes perform no jobs, or jobs are assigned to specific processes by the construction of the algorithm, independently of whether these processes participate in an execution of the algorithm. For example a solution where every process just terminates at the beginning of the algorithm, without taking any steps, is a correct solution for the at-most-once problem. Similarly, an algorithm where jobs are split into $\frac{n}{m}$ groups and each process performs the jobs in a specific group, is also a correct solution for the at-most-once problem.

Motivated by the lack of deterministic solutions with high effectiveness, we introduce in this paper the *strong at-most-once problem* and study its feasibility. The complexity measure of effectiveness [18] describes the number of jobs completed (at-most-once) by an implementation, as a function of the overall number of jobs n, the number of processes m, and the number of crashes f. The strong at-most-once problem refers to the setting where effectiveness is a function of the jobs that need to be executed and the processes that took part in the computation (took a least one step in the computation) and crashed. The strong at-most-once problem demands solutions that are adaptive, in the sense that the effectiveness depends only on the behavior of processes that participate in the execution. Specifically, if no participating process crashes in an execution, then an algorithm solving the strong at-most-once problem, will perform all tasks. In this manner trivial solutions are excluded and, as we demonstrate herein, processes have to solve an agreement primitive in order to make progress and provide a solution for the problem. In the present work, we prove that the strong at-most-once problem has *consensus number* 2 as defined by Herlihy [14]. As a result, there exists no wait-free deterministic solution for the strong at-most-once problem using atomic read/write registers only. This explains, via impossibility, the lack of deterministic solutions with high effectiveness for the at most once problem. Moreover we observe that the strong at-most-once problem belongs in the *Common*2 class as defined by Afek *et al.* [2].

Subsequently, we present a randomized k-adaptive effectiveness optimal solution for the strong at-most-once problem that has effectiveness of $n - f_k$ and expected work complexity of $O(n + k^{2+\epsilon} \log n)$ for any small constant ϵ. Sometimes k is called the contention of an execution and denotes the number of processes that *participate* in an execution of the algorithm, f_k denotes the number of participating processes that crash in an execution. Our solution is the first fully adaptive randomized solution (both in terms of effectiveness and expected work complexity) for the strong at-most-once problem. Expected Work complexity counts the expected total number of basic operations performed by the processes. Moreover our solution is anonymous, in that it does not rely on the names of processes.

Related Work: A wide range of works study at-most-once semantics in a variety of settings. At-most-once message delivery [7,21,23]and at-most-once semantics for RPC [6,22,28], are two areas that have attracted a lot of attention. Both in at-most-once message delivery and RPCs, we have two entities (sender/client and receiver/server) that communicate by message passing. Any entity may fail and recover and messages may be delayed or lost. In the first case one wants to guarantee that duplicate messages will not be accepted by the receiver, while in the case of RPCs, one wants to guarantee that the procedure called in the remote server will be invoked at-most-once [28].

In Kentros *et al.* [18], the at-most-once problem for asynchronous shared memory systems and the correctness properties to be satisfied by any solution were defined. The first wait-free deterministic algorithms that solve the at-most-once problem were provided and analyzed. Specifically they presented two algorithms that solve the at-most-once problem for two processes with optimal effectiveness and a multi-process algorithm, that employs a two-process algorithm as a building block, and solves the at-most-once problem with effectiveness $n - \log m \cdot o(n)$ and work complexity $O(n + m \log m)$. Subsequently Censor-Hillel [15] provided a probabilistic algorithm in the same setting with optimal effectiveness and expected work complexity $O(nm^2 \log m)$ by employing a probabilistic multi-valued consensus protocol as a building block. It is easy to show that the probabilistic solution of Censor-Hillel [15], is a solution for the strong at-most-once problem. Note that the solution in [15] is not k-adaptive with respect to the expected work complexity, since its work complexity depends on m and not k. Recently Kentros and Kiayias [17] presented the first wait-free deterministic algorithm for the at-most-once problem which is optimal up to additive factors of m. Specifically their effectiveness is $n - (2m - 2)$ which comes close to an additive factor of m to the known upper bound over all possible algorithms for effectiveness

$n-m+1$ (from [18]). They also demonstrate how to construct an algorithm which has effectiveness $n - \mathrm{O}(m^2 \log n \log m)$ and work complexity $\mathrm{O}(n + m^{3+\epsilon} \log n)$, and is both effectiveness and work optimal when $m = \mathrm{O}(\sqrt[3+\epsilon]{n/\log n})$, for any constant $\epsilon > 0$.

Di Crescenzo and Kiayias in [9] (and later Fitzi *et al.* [12]) demonstrate the use of the semantic in message passing systems for the purpose of secure communication. Driven by the fundamental security requirements of *one-time pad* encryption, the authors partition a common random pad among multiple communicating parties. Perfect security can be achieved only if every piece of the pad is used at most once. The authors show how the parties maintain security while maximizing efficiency by applying at-most-once semantics on pad expenditure.

Ducker *et al.* [10] consider a distributed task allocation problem, where players that communicate using a shared blackboard or an arbitrary directed communication graph, want to assign the tasks so that each task is performed exactly once. They consider synchronous execution without failures and examine the communication and round complexity required to solve the problem, providing interesting lower and upper bounds. If crashes are introduced in their model, then they will have an at-most-once version of their problem and the impossibility results from Kentros *et al.* [18] will hold.

One can also relate the at-most-once problem to the consensus problem [11,14,24,20]. Indeed, consensus can be viewed as an at-most-once distributed decision. Another related problem is process renaming, see Attiya *et al.* [5] where each process identifier should be assigned to at most one process.

The at-most-once problem has also many similarities with the Write-All problem for the shared memory model [4,8,13,16,19,25,26]. First presented by Kanellakis and Shvartsman [16], the Write-All problem is concerned with performing each task *at-least-once*. Most of the solutions for the Write-All problem, exhibit super-linear work even when $m \ll n$. Malewicz [25] was the first to present a deterministic solution for the Write-All problem that has linear work for a non-trivial number of processes (significantly less processes that the size of the Write-All problem n). The algorithm presented by Malewicz [25] has work complexity $\mathrm{O}(n + m^4 \log n)$ and uses test-and-set operations. Later Kowalski and Shvartsman [19] presented a solution for the Write-All problem that for any constant ϵ has work complexity $\mathrm{O}(n + m^{2+\epsilon})$. Their algorithm uses a collection of q permutations with contention $\mathrm{O}(q \log q)$ for a properly choose constant q. Kentros and Kiayias [17] demonstrate how to employ an iterative deterministic at-most-once algorithm in order to solve the Write-All problem with work complexity $\mathrm{O}(n + m^{3+\epsilon} \log n)$ for any constant ϵ, whithout using test-and-set operations on relying in permutations with low contention.

With respect to randomized solutions, Martel and Subramonian [26] present a randomized solution for the Write-All problem that does optimal $\mathrm{O}(n)$ work when the number of processes is less than $\frac{n}{\log n}$. Their solution assumes an oblivious adversary, which is a weaker adversary than the strong adaptive adversary we use in this work. When it comes to a strong adaptive adversary, Anderson and Woll [4] provide a $\mathrm{O}(n \log m)$ solution for $n = m^2$ write-all cells and m processes.

Contributions: We introduce the strong at-most-once problem, as the problem of solving the at-most-once problem with effectiveness that is a function of the jobs that need to be executed n and the processes that took part in the computation (took a least one step in the computation) and crashed f_k. From the upper bound on effectiveness for all algorithms (see [18]), we get that an effectiveness of $n - f_k$ for the strong at-most-once problem is optimal. We show that the strong at-most-once problem has consensus number 2 (see [14]), and thus there exists no wait-free deterministic solution for the strong at-most-once problem using read/write atomic registers. Moreover we observe that the strong at-most-once problem belongs in the *Common*2 class as defined by Afek *et al.* [2].

We present and analyze a randomized algorithm, called RA, that solves the strong at-most-once problem. The algorithm is anonymous, wait-free and k-adaptive, in the sense that both effectiveness and work complexity depend on k, the number of processes that participate in the execution. The algorithm uses randomized test-and-set as a building block, both for guaranteeing the at-most-once property and in order to facilitate the transfer of knowledge of which jobs have already been performed. Algorithm RA uses the *RatRace* algorithm from Alistarh *et. al.* [3] for the randomized test-and-set operations. It has optimal effectiveness of $n - f_k$ and expected work complexity $\mathrm{O}(n + k^{2+\epsilon} \log n)$ for any small constant ϵ. It improves over the solution of Censor-Hillel [15] which can be shown to solve the strong at-most-once problem with optimal effectiveness $n - f_k$ and expected work complexity $\mathrm{O}(nm^2 \log m)$. We note that combining the Write-All solution from Martel and Subramonian [26] with the *RatRace* algorithm from Alistarh *et. al.* may give an optimal effectiveness solution for the strong at-most-once problem with expected work complexity $\mathrm{O}(n + m \log n \log m)$ for the weaker oblivious adversary. However our proposed solution is for the stronger adaptive adversary and its expected work is k-adaptive, meaning that it depends on the number of participants k, not on the maximum number of processes m.

To our knowledge, algorithm RA provides also the first k-adaptive randomized solution for the Write-All problem.

2 Model and Definitions

2.1 Model and Adversary

We consider a system of m asynchronous, shared-memory processors in the presence of crashes. We use the *Input/ Output Automata* formalism, and specifically the *asynchronous shared memory automaton* that consists of a set of *processes* that interact by means of a collection of *shared variables* that support atomic read/write operations [24].

Each process has access to local random coin-flips. We consider a *strong, adaptive adversary* as in [3,4,15] that has complete knowledge of the algorithm executed by the processes. The adversary controls asynchrony and crashes. This is modeled by allowing the adversary to make all scheduling decisions. The adversary can base its next scheduling decisions on the local state of all the processes, including the results of local coin-flips. Notice, however, that the adversary does not know the results of local coins that were not yet flipped. The adversary can cause $m - k$ ($k > 1$) crashes at the beginning of the execution (before processes take any steps). From the k participating processes $f_k < k$ can crash during the execution, after the process that crashes takes at least one step. The total number of crashes allowed is $f = (m - k) + f_k < m$. We denote by $\mathit{fairexecs}_f(A)$ all fair executions of A with f crashes and by $\mathit{fairexecs}_{f,f_k}(A)$, where $f \geq f_k$, all fair executions of A where $k > f_k$ processes take at least one step in the execution and exactly f_k of the k processes crash.

Note that since the processes can only communicate by accessing atomic read/write operations in the shared memory, all the asynchronous executions are linearizable. This means that concurrent actions can be mapped to an equivalent sequence of transitions, where only one process performs an action in each transition, and thus the model presented above is appropriate for the analysis of concurrent multi-process systems.

2.2 Problem Definitions and Metrics

We consider algorithms that perform a set of tasks, called *jobs*. Let A be an algorithm specified for m processes from set $\mathcal{P}$, and for n jobs with unique ids from set $\mathcal{J} = [1 \ldots n]$. We assume that there are at least as many jobs as there are processes, i.e., $n \geq m$. We model the performance of job j by process p by means of action $\mathsf{do}_{p,j}$. For a sequence c, we let $len(c)$ denote its length, and we let $c|_\pi$ denote the sequence of elements π occurring in c. Then for an execution α, $len\left(\alpha|_{\mathsf{do}_{p,j}}\right)$ is the number of times process p performs job j. Finally we denote by $F_\alpha = \{p | stop_p \text{ occurs in } \alpha\}$ the set of crashed processes in execution α. We now define the number of jobs performed in an execution, the *at-most-once problem* and *effectiveness* (from Kentros *et al.* [18]).

Definition 1. *For execution α let $\mathcal{J}_\alpha = \{j \in \mathcal{J} | \mathsf{do}_{p,j}$ occurs in α for some $p \in \mathcal{P}\}$. The total number of jobs performed in α is defined to be $Do(\alpha) = |\mathcal{J}_\alpha|$.*

Definition 2. *Algorithm A solves the at-most-once problem if for each execution α of A we have $\forall j \in \mathcal{J} : \sum_{p \in \mathcal{P}} len\left(\alpha|_{\mathsf{do}_{p,j}}\right) \leq 1$.*

Definition 3. *$E_A(n, m, f) = \min_{\alpha \in \mathit{fairexecs}_f(A)}(Do(\alpha))$ is the **effectiveness** of algorithm A , where m is the number of processes, n is the number of jobs, and f is the number of crashes. An alternate definition is $E_A(n, m, f, f_k) = \min_{\alpha \in \mathit{fairexecs}_{f,f_k}(A)}(Do(\alpha))$, where $f_k \leq f$ the number of processes that crashed in an execution after taking at least one step.*

A trivial algorithm can solve the at-most-once problem by splitting the n jobs in groups of size $\frac{n}{m}$ and assigning one group to each process. Such a solution has effectiveness $E(n, m, f) = (m - f) \cdot \frac{n}{m}$ (consider an execution where f processes fail at the beginning of the execution).

We also define the *strong at-most-once problem.* The strong at-most-once problem, requires that only processes that participate in an execution and fail can block an at-most-once job. Alternatively, the strong at-most-once problem requires that all jobs are performed if no participating processes fail ($f_k = 0$). Trivial solutions for the at-most-once problem, such as the one described above are not valid solutions for the strong at-most-once problem. We show that the strong at-most-once problem has consensus number 2 as defined in [14] and belongs in the Common2 class of objects as defined in [2].

Definition 4. *Algorithm A solves the* ***strong at-most-once problem*** *if algorithm A solves the at-most-once problem and there exists function* $\varphi()$*, such that* $\varphi(0) = 0$ *and for all* f, f_k*, with* $m > f \geq f_k$*,* $E_A(n, m, f, f_k) = n - \varphi(f_k)$*.*

The difference between the at-most-once problem and the strong at-most-once problem is that the latter requires that algorithms are implemented, such that in all initial states of the algorithm, no job is preassigned to a process. In other words, no process can start by performing a job, without first getting information about the current state of the execution. Moreover any job, may be performed by any process in some execution of the algorithm. In that sense, the 2-process effectiveness optimal algorithm from Kentros et. al. [18], is not a solution for the strong at-most-once problem, since the job with id 1 (resp. with id n) cannot be performed by the process with pid 1 (resp. pid 0).

Expected work complexity measures the expected (over the coin flips of the processes) total number of basic operations (comparisons, additions, multiplications, shared memory reads and writes) performed by an algorithm. We assume that each internal or shared memory cell has size $O(\log n)$ bits and performing operations involving a constant number of memory cells costs $O(1)$. This is consistent with the way work complexity is measured in previous related work [16,19,25]. We are interested in k-adaptive algorithms and thus we want the expected work complexity to be expressed as a function of n the total number of jobs and k the number of processes that participate in an execution.

Definition 5. *The* ***expected work*** *of algorithm A, denoted by* W_A*, is the expected total number of basic operations performed by all the processes in A.*

We will prove that the strong at-most-once problem has consensus number 2. Informally a *consensus protocol* is a system of n processes that communicate through a set of shared objects. Each process starts with an input value. Processes communicate with one another by applying operations to the shared objects and eventually agree on a common input value and halt. A consensus protocol is required to be a) consistent: distinct processes never decide on distinct values, b) wait-free: each non-failed process decides after a finite number of steps, c) valid: the common decision value is the input to some process (from Herlihy [14]). We say that an object X *solves* n*-process consensus*, if there exist a consensus protocol for n-process that uses a set of objects X and read/write registers, where X can be initialized in any state. We provide the definition of consensus number form Herlihy [14].

Definition 6. *The consensus number for* X *is the largest* n *for which* X *solves* n*-process consensus. If no largest* n *exists, the consensus number is said to be infinite.*

Moreover we observe that the strong at-most-once problem belongs in the *Common*2 class of objects as defined by Afek *et al.* [2], since a wait-free implementation for the m process strong at-most-once problem can be constructed from read/write registers and 2-consensus objects (test-and-set). It is easy to see that the strong at-most-once problem belongs in Common2 even in the unbounded concurrency model of [1,27].

Finally we repeat here the following upper bound on *effectiveness* from Kentros et al. [18].

Theorem 1. *[18] For all algorithms A that solve the at-most-once problem with m processes and $n \geq m$ jobs in the presence of $f < m$ crashes it holds that $E_A(n, m, f) \leq n - f$.*

From the upper bound we have that solutions for the strong at-most-once problem with effectiveness of $n - f_k$ are optimal.

3 Consensus Number and Common2

In this section, we show that the strong at-most-once problem has consensus number 2 and belongs in Common2. As a result, we have from [14] that there exists no wait-free deterministic algorithm that solves the strong at-most-once problem, using only atomic read/write registers. Current deterministic solutions for the at-most-once, as presented in [17,18], use only atomic read/write registers and are wait-free, thus they do not offer a solution for the strong at-most-once problem.

We need to prove that the strong at-most-once problem has consensus number at least 2 (Lemma 1). The intuition behind the proof is the following; because the effectiveness of strong at-most-once solutions is $n - \phi(f_k)$, where $\phi(0) = 0$, a job is not completed if and only if a process that participated in the execution of the strong at-most-once algorithm crashes. If no process crashes, the job is performed and this can break symmetry. If one of the processes crashes, the other will not (at least one process does not crash). Moreover the fact that some job has not been performed by process p, reveals that the other process has participated in the protocol. Process p can use this knowledge in order to safely decide some value. To show that the strong at-most-once problem has consensus number at most 2, we construct a solution for the strong at-most-once problem using test-and-set operations.

Lemma 1. *The strong at-most-once problem has consensus number at least* 2*.*

Proof. In order to prove that the strong at-most-once problem has consensus number at least 2, we assume that there exists a wait-free algorithm A that solves the strong at-most-once problem for 2 processes. We demonstrate how to implement a wait-free algorithm A' that solves consensus for 2 processes, using algorithm A and atomic read/write shared memory registers. Since the nature of the jobs performed by algorithm A are external to the problem and the algorithm, we treat algorithm A as a black box implementation, and define

the jobs algorithm A is performing so that when process p performs job i, it writes its identifier p in the $i-$th position of an array W. Algorithm A' works for 2 processes p, q and uses an array C of size 2 and an array W of size n where n the size of the strong at-most-once problem algorithm A solves. Array W has its cells initialized to the $\perp$ value. Since algorithm A solves the at-most-once problem only one process may write its identifier in any single position of the array W. Algorithm A' works as follows:
Process p writes its proposed input value in position p of the array C and then invokes A. After process p terminates the execution of A, process p reads the value stored in position 0 of the array W. If the value it reads is its process identifier, it decides the value it proposed, otherwise it decides on the value proposed by the other process that participates in algorithm A'.

Since algorithm A is wait-free A' is also wait-free. Now we need to prove consistency and validity, namely that distinct processes never decide on distinct values and that the common decision value is the input to some process.

For process p we have 2 cases, process p either decides the value stored in C_p, or the value stored in C_q. Case 1: Process p decides C_p. The value it decides can only be the input value of process p, since from A' only p may write in C_p and process p first writes its input value in C_p, then participates in algorithm A and from the outcome of A, it either reads C_p or C_q and decides on the value it read. Since process p reads C_p it follows that it reads in position W_0 its process id, and thus p performed job 0. Moreover if process q participated in A, q did not perform job 0 (at-most-once property). So from A' it follows that q decided on C_p. Now we only need to prove that the value q read in C_p is the input of p. If q read a value different than the input of p, it follows that when q returned from the invocation of A, p had not yet written its input value in C_p, and consequently had not invoked A. This is a contradiction, since A solves the strong at-most-once problem. If process q is the only process invoking A and q does not fail, then the effectiveness of A should be n, which implies that process q performed job 0, a contradiction.

Case 2: Process p decides C_q. This means that process p invoked A, algorithm A returned and process p did not perform job 0. Since A solves the strong at-most-once problem, it means that process q invoked A before process p terminated (otherwise process p would be executing A alone and thus p should have performed all the at-most-once jobs), which from A' implies that C_q contains the input value of process q. Process q either terminates A without crashing, or crashes before completing A. In the first case, since process p terminated and did not crash while executing algorithm A, we have that A should have effectiveness n, which implies that process q executed job 0, and if process q decides, it decides on the value of C_q, the input of q. Otherwise process q has crashed and p is the only process that decides the input of q. This completes the proof.

It is easy to see that by associating each at-most-once job with one test-and-set object and having the process that succeeds in the test-and-set perform the at-most-once job before accessing a new test-and-set, we have an effectiveness optimal strong at-most-once solution for an unbounded number of processes using only read/write registers and consensus 2 objects.

Theorem 2. *The strong at-most-once problem has consensus number* 2.

Corollary 1. *The strong at-most-once problem belongs in the Common*2 *class of objects.*

4 Algorithm RA

In Algorithm RA (Fig. 1) jobs are grouped in super-jobs. Each super-job contains $\log n$ at-most-once jobs. Every job is associated with a shared memory element of matrix W. The matrix W has size $\frac{n}{\log n}$ x $(\log n + 1)$. The row 0 of matrix W is associated with the $\frac{n}{\log n}$ super-jobs, while the rows $\{1, \ldots, \log n\}$ are associated with the n jobs. Super-job i consists of the $\log n$ jobs that are associated with elements $W[i][j]$ for all $j \in \{1, \ldots, \log n\}$. Each element of the matrix W supports a randomized wait-free atomic test-and-set operation. The job test-and-set operations are used in order to guarantee at-most-once semantics while the super-jobs test-and-set operations are used to detect and resolve collisions between processes.

Processes in Algorithm RA create intervals of super-jobs. The main idea is that every process p picks a random super-job i as a candidate starting point for its interval. Process p calls all the test-and-set operations related with the jobs grouped under i. For each test-and-set operation p wins, it performs the

```
RA for process $p$:
FREE $\leftarrow \{0, \ldots, \frac{n}{\log n} - 1\}$
$size \leftarrow \frac{n}{\log n}$
while($size > 0$)
  $next \leftarrow$ FREE.$random()$
  for($i \leftarrow 1, i \leq \log n, i++$)
    if W[next][i].tas() then
      $j = next \cdot \log n + i$
      $\mathsf{do}_{j,p}$
    endif
  endfor
  $flag \leftarrow W[next][0].tas()$
  if($flag$) then
    $head \leftarrow next$
    $tail \leftarrow next$
    while($flag$)
      $W[next][0].setHead(head)$
      $W[head][0].setTail(tail)$
      $next++$
      if($next < \frac{n}{\log n}$)then
        $tail \leftarrow next$
        for($i \leftarrow 1, i \leq \log n, i++$)
          if W[next][i].tas() then
            $j = next \cdot \log n + i$
            $\mathsf{do}_{j,p}$
          endif
        endfor
        $flag \leftarrow W[next][0].tas()$
      else
        $flag \leftarrow FALSE$
      endif
    endwhile
    $size \leftarrow$ FREE.$remove(head, tail)$
  else
    $head \leftarrow next$
    $tail \leftarrow next$
    $tmp \leftarrow W[next][0].getHead()$
    if($tmp \; != \perp$)&&($tmp < head$) then
      $head \leftarrow tmp$
    endif
    $tmp \leftarrow W[head][0].getTail()$
    if($tmp \; != \perp$)&&($tmp > tail$) then
      $tail \leftarrow tmp$
    endif
    $size \leftarrow$ FREE.$remove(head, tail)$
  endif
endwhile
```

Fig. 1. Algorithm RA: pseudocode

corresponding job. When done, it calls the test-and-set operation of the super-job i. If the procedure described above takes place in an execution α of algorithm RA, we say that process p has *performed* super-job i, or that super-job i has been *performed*. This is independent of whether process p won the test-and-set operation for super-job i. If there exists no process p that has performed super-job i in execution α, we say that the super-job i is still *available* in α.

If process p wins the test-and-set operation for super-job i, it marks i as the starting point of its working interval, then performs super-job $i + 1$ and keeps moving, one super-job at a time in a rightward direction, until it loses a super-job test-and-set operation. As long as p wins test-and-set operations on the super-jobs it performs, it adds the super-jobs to its current interval. If it fails this means that some other process has marked the specific super-job as the beginning of its working interval. In this case, p stops working on the interval and picks randomly a new super-job in order to start a new working interval.

The key idea behind this approach is that a process p keeps expanding a working interval that started at some super-job i until it loses a test-and-set operation at some super-job j $(j > i)$. This means that some other process q won that test-and-set for j and thus q continues expanding the interval that started at i. This leads to the observation that if k processes are participating in an execution of the algorithm RA, there exist at most $k + 1$ intervals of performed super-jobs, from which, at any given point of the execution, at most k are being expanded from the right end. As long as the available super-jobs are significantly more than the k processes participating in an execution of the algorithm, we can show from the above discussion (see Section 5) that processes that need a new random starting point, will likely be positioned far enough from the endpoints of the existing intervals of performed super-jobs. This results in substantial progress being done before a process working on an interval loses a test-and-set operation and thus has to start a new interval. The latter will allow us to show that if a process has an outdated estimation of the available super-jobs, by colliding with large intervals of performed super-jobs, it will be able to learn fast about super-jobs that have been completed.

Next we present the shared memory variables, internal variables and the steps a process p has to take in algorithm RA in more detail:

Shared Variables. Algorithm RA uses matrix W of size $\frac{n}{\log n}$ x $(\log n + 1)$. The matrix is stored in shared memory. Each element of the matrix is initially set to 0 and supports, through function $tas()$, a randomized wait-free test-and-set operation. The $tas()$ function, when invoked on element $W[i][j]$ of the matrix, tests if $W[i][j]$ is 0 and sets $W[i][j]$ to 1. If element $W[i][j]$ is 0, function $W[i][j].tas()$ returns $TRUE$ and we say that the process p that called $W[i][j].tas()$ *wins* or *succeeds* in the test-and-set operation. If element $W[i][j]$ is 1, function $W[i][j].tas()$ returns $FALSE$ and we say that the process p that called $W[i][j].tas()$ loses the test-and-set operation. There are various randomized implementations for test-and-set on asynchronous shared memory. We use the *RatRace* algorithm from Alistarh *et. al.* [3]. Since RatRace is anonymous and RA does not rely on process identifiers, RA is also anonymous.

Each of the n elements of the rows $\{1, \ldots, \log n\}$ of matrix W corresponds to one at-most-once job. Moreover, each of the $\frac{n}{\log n}$ elements of row 0 of matrix W is associated with a super-job of $\log n$ jobs. In addition, each element of row 0 has two pointers: *head* and *tail*. The pointers are initially set to $\perp$. When their value is different from $\perp$ they point to elements of row 0. These elements correspond to the beginning and the end (respectively) of an interval of super-jobs that some process p has been working on. A process may access pointers *head* and *tail* through the $getHead()$, $getHead()$, $setTail()$ and $getTail()$ functions. Note that in RA an element of row 0 of matrix W is only set to 1 through an invocation of the test-and-set function associated with it. Moreover, if $W[i][0]$ is set to 1 for some $i \in \{0, \ldots, \frac{n}{\log n} - 1\}$, then for all $j \in \{1, \ldots, \log n\}$ we have that $W[i][j]$ is set to 1 through an invocation of the $W[i][j].tas()$ test-and-set function.

Internal Variables of Process p. The variable FREE keeps the set of super-jobs that process p has not verified as performed. The variable FREE is a tree structure that keeps track of intervals of available super-jobs. Each interval of available jobs is stored in a leaf node as two pointers (beginning and ending of the interval). Initially the FREE set has only 1 leaf node that contains two pointers to 0 and $\frac{n}{\log n} - 1$. Process p interacts with FREE using 2 functions. The function *random*() returns an element from the FREE set uniformly at random. Since the set FREE is stored in a tree structure, retrieval of a random element can be done in $O(\log l)$ where l the number of leaves of the tree. The function *remove*(*head*, *tail*) removes from the set FREE the interval of elements beginning in *head* and ending in *tail*, or the subset of elements of the interval $\{head, \ldots, tail\}$ that intersects with the set FREE. The function returns as output the number of elements left in set FREE.

The variable *size* stores the number of elements in the set FREE. It is initially set to $\frac{n}{\log n}$ and it is only updated when the function *remove*() is called on variable FREE.

The variables *head* and *tail* hold the endpoints of the current working interval of super-jobs of process p or the endpoints of the interval of super-jobs that process p has learned to have been performed.

The variable *next* holds the next super-job process p is performing.

The variable *tmp* holds values of pointers fetched from the shared memory through the $getHead()$ or $getTail()$ functions.

The variable $flag$ normally holds the output of the latest test-and-set function called on a location of the 0 row and is used for exiting the inner *while* loop.

Finally variable i is used as index in for loops and j as index for at-most-once jobs, in the `do` action.

Description of Algorithm RA **for Process p.** Initially process p sets the FREE set to contain all super-jobs and sets the *size* variable to $\frac{n}{\log n}$ which is the number of super-jobs. As long as there are more super-jobs to perform ($size > 0$) process p executes the following:

Process p picks a super-job to perform uniformly at random from set FREE. For each job grouped under the selected super-job, p calls the job's test-and-set operation and performs the job if it wins the associated test-and-set operation. Independently of whether the test-and-set operation was successful, p proceeds to the next job until all the test-and-set operations associated with the jobs under the super-job have been called.When all the test-and-set operations have been called, process p calls the test-and-set operation associated with the selected super-job.

If process p wins the test-and-set operation on the randomly selected super-job, process p starts working on an interval of super-jobs. The interval starts at the randomly selected super-job and moves to the right. Process p peforms one super-job at a time,moving rightwards. As long as process p wins the test-and-set operations associated with the super-jobs, p keeps expanding the interval from the right side.The interval ends when the first super-job test-and-set operation fails. When this happens, process p removes the interval from the FREE set and as long as there are still super-jobs left in set FREE, picks a new random starting point and repeats the process. Note also that process p keeps the shared memory updated with the interval it is currently working at (or has just finished working), through the *head* and *tail* pointers of the row 0 of the W matrix.

If the test-and-set operation on the randomly selected super-job fails, process p reads the *head* pointer of the respective shared memory location and if it contains a valid value, it reads the *tail* pointer from the $W[head][0]$ location of the shared memory. Then it removes from the FREE set, the interval indicated by the *head* and *tail* pointers, and picks a new super-job at random. Essentially process p detects that it collided in the specific super-job with another process and attempts to increase the knowledge it has about jobs that have already been performed, by learning the interval of super-jobs that have been completed around the position of collision.

5 Analysis of Algorithm RA

Here we provide the intuition behind the proofs of correctness, effectiveness and expected work complexity.

In order to show that algorithm RA solves the strong at-most-once problem, we prove that RA solves the at-most-once problem (no job is executed more than once), it has effectiveness $n - f_k$ and is wait-free. The first two parts are straight forward. Based on the correctness properties of the test-and-set operations, it is easy to see that a job cannot be performed more than once. Moreover, in order to prove effectiveness of $n - f_k$, we need to observe that in RA, a job is performed, unless some process p wins the test-and-set operation associated with it, and then crashes before it performs the job. After this observation, we only need to show that all test-and-set operations are invoked by some process before RA terminates, and that after winning a test-and-set operation, a process does not call another test-and-set operation before it completes the job associated with the test-and-set it has already won. We still need to show the wait-free property. The proof is based on the observation that every time the main loop is executed by process p, at least one element will be removed from the local FREE set of

p. The set has a finite number of elements and thus process p has to terminate if it does not fail. Following the strategy above we can prove Theorem 3.

Theorem 3. *Algorithm* RA *is wait-free and solves the strong at-most-once problem with effectiveness* $n - f_k$.

In order to prove that the expected work complexity of the algorithm is $O(n + k^{2+\epsilon} \log n)$, for any constant ϵ, we use the following strategy. We first observe that at any point of the execution, the set of available super-jobs is split into at most $k+1$ intervals. We then show that when a process samples a new super-job i to start a new interval; if i belongs to the set of available super-jobs then i belongs to one of the $k+1$ intervals and the expected distance separating i from the endpoints of the intervals is large enough to allow significant progress before the process needs to sample again. Using the same strategy, we can also show that when a process stops working in an interval of super-jobs, the expected size of the completed interval it leaves behind is significant. Based on the above, we prove that whenever the process samples an element, it is either able to perform significant work before it needs to sample again or it updates its knowledge about performed super-jobs with significant information. Using the above, we first show that for performing $n - k^2$ super-jobs and learning that those $super - jobs$ have been performed, algorithm RA needs expected work $O(n)$. At this point k^2 super-jobs still need to be performed. Next we show that for any constant ϵ, RA needs expected work $O(k^{1+\epsilon} \log n)$ in order to perform the next $k^2 - k^{1+\epsilon}$ super-jobs. It is easy to see that for the last $k^{1+\epsilon}$ super-jobs, RA needs expected work $O(k^{1+\epsilon} \log n)$. With the above strategy, we get the expected work complexity in Theorem 4.

Theorem 4. *Algorithm* RA *has expected work complexity* $O(n + k^{2+\epsilon} \log n)$ *for any constant* ϵ.

If in algorithm RA we replace the job test-and-set operations with Write-All cells, RA solves the Write-All problem with work complexity $O(n + k^{2+\epsilon} \log n)$.

References

1. Afek, Y., Gafni, E., Morrison, A.: Common2 extended to stacks and unbounded concurrency. In: PODC, pp. 218–227. ACM (2006)
2. Afek, Y., Weisberger, E., Weisman, H.: A completeness theorem for a class of synchronization objects. In: PODC, pp. 159–170. ACM (1993)
3. Alistarh, D., Attiya, H., Gilbert, S., Giurgiu, A., Guerraoui, R.: Fast Randomized Test-and-Set and Renaming. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 94–108. Springer, Heidelberg (2010)
4. Anderson, R.J., Woll, H.: Algorithms for the certified write-all problem. SIAM J. Computing 26(5), 1277–1283 (1997)
5. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. J. ACM 37(3), 524–548 (1990)
6. Birrell, A.D., Nelson, B.J.: Implementing remote procedure calls. ACM Trans. Comput. Syst. 2(1), 39–59 (1984)

7. Chaudhuri, S., Coan, B.A., Welch, J.L.: Using adaptive timeouts to achieve at-most-once message delivery. Distrib. Comput. 9(3), 109–117 (1995)
8. Chlebus, B.S., Kowalski, D.R.: Cooperative asynchronous update of shared memory. In: STOC, pp. 733–739 (2005)
9. Di Crescenzo, G., Kiayias, A.: Asynchronous Perfectly Secure Communication over One-Time Pads. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 216–227. Springer, Heidelberg (2005)
10. Drucker, A., Kuhn, F., Oshman, R.: The communication complexity of distributed task allocation. In: PODC, pp. 67–76. ACM (2012)
11. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)
12. Fitzi, M., Nielsen, J.B., Wolf, S.: How to share a key. In: Allerton Conference on Communication, Control, and Computing (2007)
13. Groote, J., Hesselink, W., Mauw, S., Vermeulen, R.: An algorithm for the asynchronous write-all problem based on process collision. Distributed Computing 14(2), 75–81 (2001)
14. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13, 124–149 (1991)
15. Hillel, K.C.: Multi-sided shared coins and randomized set-agreement. In: Proc. of the 22nd ACM Symp. on Parallel Algorithms and Architectures (SPAA 2010), pp. 60–68 (2010)
16. Kanellakis, P.C., Shvartsman, A.A.: Fault-Tolerant Parallel Computaion. Kluwer Academic Publishers (1997)
17. Kentros, S., Kiayias, A.: Solving the At-Most-Once Problem with Nearly Optimal Effectiveness. In: Bononi, L., Datta, A.K., Devismes, S., Misra, A. (eds.) ICDCN 2012. LNCS, vol. 7129, pp. 122–137. Springer, Heidelberg (2012)
18. Kentros, S., Kiayias, A., Nicolaou, N., Shvartsman, A.A.: At-Most-Once Semantics in Asynchronous Shared Memory. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 258–273. Springer, Heidelberg (2009)
19. Kowalski, D.R., Shvartsman, A.A.: Writing-all deterministically and optimally using a nontrivial number of asynchronous processors. ACM Transactions on Algorithms 4(3) (2008)
20. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16(2), 133–169 (1998)
21. Lampson, B.W., Lynch, N.A., S-Andersen, J.F.: Correctness of at-most-once message delivery protocols. In: FORTE, pp. 385–400 (1993)
22. Lin, K.-J., Gannon, J.D.: Atomic remote procedure call. IEEE Trans. Softw. Eng. 11(10), 1126–1135 (1985)
23. Liskov, B., Shrira, L., Wroclawski, J.: Efficient at-most-once messages based on synchronized clocks. ACM Trans. Comput. Syst. 9(2), 125–142 (1991)
24. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers (1996)
25. Malewicz, G.: A work-optimal deterministic algorithm for the certified write-all problem with a nontrivial number of asynchronous processors. SIAM J. Comput. 34(4), 993–1024 (2005)
26. Martel, C., Subramonian, R.: On the complexity of certified write-all algorithms. J. Algorithms 16, 361–387 (1994)
27. Merritt, M., Taubenfeld, G.: Computing with Infinitely Many Processes. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 164–178. Springer, Heidelberg (2000)
28. Spector, A.Z.: Performing remote operations efficiently on a local computer network. Commun. ACM 25(4), 246–260 (1982)

Brief Announcement: Wait-Free Gathering of Mobile Robots

Zohir Bouzid[1], Shantanu Das[2], and Sébastien Tixeuil[1]

[1] University Pierre et Marie Curie - Paris 6, LIP6-CNRS 7606, France
[2] Ben-Gurion University & Technion - Israel Institute of Technology, Israel

Robot Systems. This paper considers distributed systems of autonomous robots that can move freely on the two-dimensional Euclidean space, have visibility sensors (to see other robots, obstacles etc.) and can perform computations. One of the fundamental problems in distributed coordination of robots is to gather the robots at a single location. The gathering problem has been studied under various models with the objective of determining the minimal set of assumptions that still allows the robots to gather successfully within a finite time. For example, it is known that gathering can be solved even if the robots are *anonymous* (indistinguishable from each-other), *oblivious* (no persistent memory of the past), and cannot communicate explicitly with each other (except for indirect signaling using movement). Further, the robots may not share a common sense of direction. Robots operate in *cycles* that comprise *look*, *compute*, and *move* phases. The look phase consists in taking a snapshot of the other robots positions. In the compute phase, a robot computes a target destination, based on the previous observation, using a deterministic algorithm and in the move phase, the robot moves toward the computed destination (although the move may end before reaching the target destination). We consider the semi-synchronous ATOM model [4], where each cycle is considered to be atomic but only a subset of the robots may be active in each cycle. The robots are modeled as points on the Euclidean plane and the objective is to gather all robots at a single point.

In this model, the gathering problem can be solved for any $n > 2$ robots starting from distinct locations and cannot be solved deterministically for $n = 2$ robots [4]. Since the robots are assumed to be oblivious, the algorithms are robust against memory corruption. However, none of the known solutions for gathering of robots consider more severe and permanent faults, except for the seminal results of Agmon and Peleg [1] who solved gathering in the presence of crash faults (i.e. when some of the robots may arbitrarily stop moving). That paper provided an algorithm for gathering all non-faulty robots when at most one robot may crash, assuming that the robots are initially in distinct locations. On the other hand, Dieudonné and Petit [3] solved gathering starting from arbitrary configuration of robots (possibly with multiple robots at the same location), while assuming that there are an odd number of robots and no robot fails. We consider the gathering problem in presence multiple crash faults and starting from any arbitrary configuration of robots. Note that if the robots are initially located in exactly two distinct points with an equal number of robots in each location (we call this the *bivalent* configuration) then gathering is not possible even in the fault-free case[1]. In fact, this is the only scenario where gathering is not possible and we show how to solve gathering (of correct robots) starting from any other configuration, even if up to $f < n$ robots crash. To achieve these results, we introduce the additional capability of *strong* multiplicity detection (i.e. our robots can sense the exact number of robots at any point, unlike in [1,4]) and chirality (i.e. our robots share a common handedness). The former capability is essential for gathering from arbitrary configurations, while it not know whether the latter capability is also necessary.

Symmetric Configurations and Weber Points. Given any set of points C on the plane, a *Weber-point* minimizes the sum of distances from the points in C. The Weber-point is unique for any non-collinear set of points and remains invariant under the movement of any point in C towards it. However it is difficult to compute such a point and only for certain configurations that are *symmetric* or *regular*, there exist algorithms to compute the Weber-point. In this paper we define a larger class of configurations called *quasi-regular*, and provide algorithms to (i) detect if a given configuration C is *quasi-regular* and (ii) compute the Weber-point of any configuration that is *quasi-regular* and not collinear.

Gathering Algorithm. Our algorithm classifies the configuration of robots at any time to one of the following types: $\mathcal{B}, \mathcal{M}, \mathcal{L}, \mathcal{QR}, \mathcal{A}$. A configuration of type $\mathcal{M}$ contains a point u whose multiplicity is greater than that of any other robot location and all robots simply move to u (while avoiding the creation of another point of multiplicity). For any collinear ($\mathcal{L}$) or quasi-regular ($\mathcal{QR}$) configuration that has a unique Weber-point, the robots move towards that point. For collinear configurations that do not have a unique Weber-point, the robots move to form either a non-collinear configuration or a type $\mathcal{M}$ configuration. All other configurations that are not $\mathcal{B}$, $\mathcal{M}$, $\mathcal{L}$, or, $\mathcal{QR}$, must be asymmetric (of type $\mathcal{A}$) and in such configurations, it is possible to uniquely order the robot locations. The robots select one of these locations u which is also a *safe* point (i.e. movement towards u may never create a bivalent configuration) and the robots move towards this point u. We show that starting from any configuration that is not bivalent ($\mathcal{B}$), the algorithm succeeds in gathering all non-faulty robots at a single location within a finite time. The complete algorithm and proof of correctness can be found in the full version of the paper [2].

References

1. Agmon, N., Peleg, D.: Fault-tolerant gathering algorithms for autonomous mobile robots. SIAM Journal of Computing 36(1) (2006)
2. Bouzid, Z., Das, S., Tixeuil, S.: Wait-Free Gathering of Mobile Robots. ArXiv e-prints:1207.0226 (2012)
3. Dieudonné, Y., Petit, F.: Self-stabilizing Deterministic Gathering. In: Dolev, S. (ed.) ALGOSENSORS 2009. LNCS, vol. 5804, pp. 230–241. Springer, Heidelberg (2009)
4. Suzuki, I., Yamashita, M.: Distributed anonymous mobile robots: Formation of geometric patterns. SIAM Journal of Computing 28(4), 1347–1363 (1999)

[1] This is a simple extension of the impossibility result from [4].

Brief Announcement: Distributed Exclusive and Perpetual Tree Searching[⋆]

Lélia Blin[1,⋆⋆], Janna Burman[2], and Nicolas Nisse[3,⋆⋆⋆]

[1] Univ. d'Evry Val d'Essone, et LIP6, Univ. Pierre et Marie Curie, Paris, France
[2] Grand Large, INRIA, LRI, Orsay, France
[3] MASCOTTE, INRIA, I3S(CNRS/UNS), Sophia Antipolis, France

Abstract. We tackle a version of the well known *graph searching* problem where a team of robots aims at capturing an intruder in a graph. The robots and the intruder move between the graph nodes. The intruder is invisible, arbitrary fast, and omniscient. It is caught whenever it stands on a node occupied by a robot, and cannot escape to a neighboring node. We study graph searching in the CORDA model of mobile computing: robots are asynchronous and perform cycles of *Look-Compute-Move* actions. Moreover, motivated by physical constraints and similarly to some previous works, we assume the *exclusivity* property, stating that no two or more robots can occupy the same node at the same time. In addition, we assume that the network and the robots are anonymous. Finally, robots are *oblivious*, i.e., each robot performs its move actions based only on its current "vision" of the positions of the other robots. Our objective is to characterize, for a graph G, a set of integers such that for every integer k in the set, *perpetual* graph searching can be achieved by a team of k robots starting from *any* k distinct nodes in G. One of our main results is a full characterization of this set, for any asymmetric tree. Towards providing a characterization for all trees, including trees with non-trivial automorphisms, we have also provided a set of positive and negative results, including a full characterization for any line. All our positive results are based on the design of graph searching algorithms.

1 Introduction

This BA announces a work that aims at understanding the algorithmic power and limitation of computing with autonomous mobile robots. The literature dealing with this objective has considered different kinds of coordination tasks, including pattern formation, robot gathering, and graph exploration. Each of these tasks involves complex coordination protocols for the robots, whose complexity depends on the capabilities of the robots in terms of perception of their environment, individual computational power, and communication. The CORDA model has been introduced for capturing the essence of mobile computing. It focusses on the asynchronous nature of the actions performed by the robots, and on the limitations

[⋆] Based on: L. Blin, J. Burman, and N. Nisse: Perpetual Graph Searching, Research Report INRIA-RR-7897, Sophia-Antipolis, France, Feb. 2012.

[⋆⋆] Additional support from the ANR project "SHAMAN".

[⋆⋆⋆] Partially supported by FP7 STREP EULER (N.N.).

caused by the absence of communication between them. In this model, robots are endowed with visibility sensors allowing each robot to perceive the positions of all the other robots. They operate in asynchronous *Look-Compute-Move* action cycles. During its *look* action, a robot perceives the relative positions of the other robots; during the *compute* action, it executes some individual deterministic computation whose input is the set of the latest perceived positions of the other robots (which may have changed since they were measured); finally, during the *move* action, the robot changes its position according to its computation.

The coordination of autonomous deterministic mobile robots has been first studied in *continuous* environments (e.g., the 2-dimensional Euclidean space). In the discrete CORDA model, it is not clear whether even simple coordination tasks can be achieved. As a consequence, most of the literature on robot computing assume additional hypotheses, including presence of identities enabling to distinguish robots, capacity to store the sequence of all previous perceived positions and moves, sense of direction, and the ability to construct *towers* of robots (i.e., to bring several robots at a node). We have proved that a very complex task, like *graph searching*, can be solved without any of these assumptions. We assume that the robots have no identities, they are memoryless (i.e., stateless), have no sense of direction, and two or more robots cannot occupy the same position at the same time (*exclusivity* property). We call *min*-CORDA this essential CORDA model.

2 Contributions

For any graph G, let $\mathsf{fs}(G)$ denote the set of robot team sizes such that $k \in \mathsf{fs}(G)$ if and only if distributed graph searching in G can be achieved by a team of k robots starting from *any* k (distinct) nodes. Our main result consists in a characterization of $\mathsf{fs}(T)$, for any asymmetric tree T. Let $\mathsf{xs}(T)$ denote the *exclusive search number*, i.e., the minimum number of robots to be used for capturing the intruder in a *centralized* setting satisfying the exclusivity property. By definition, $k \in \mathsf{fs}(T)$ implies that $k \geq \mathsf{xs}(T)$. We have proved that, when T possesses no symmetries (i.e., has no non-trivial automorphisms), then $k \in \mathsf{fs}(T)$ for all $k \geq \mathsf{xs}(T) + 1$. This result is based on the explicit design of a distributed protocol enabling perpetual graph searching by k robots, for any $k \geq \mathsf{xs}(T) + 1$.

When T possesses symmetries, the computation of $\mathsf{fs}(T)$ becomes more complex. In this case, we have shown that $\mathsf{fs}(T)$ depends on the set $\mathcal{S}_T$ of isomorphic nodes of T separated by a path of even length. In particular, $k \notin \mathsf{fs}(T)$ for any $k \in [|\mathcal{S}_T|, |\mathcal{S}_T| + \mathsf{xs}(T_0)]$, where T_0 results from T after removing all nodes appearing in $\mathcal{S}_T$. Our impossibility results are not based on the perpetual nature of graph searching in min-CORDA, but on the exclusivity property. For the simpler case of a line, we have fully characterized $\mathsf{fs}(L)$ for any n-node line L. All our positive results are constructive.

We note that *exclusive* graph searching behaves very different from the classical one. As a result and due to min-CORDA, the proofs and the algorithms we propose are very different and more involved than in the classical case. Our proofs introduce several techniques that may prove useful also in future studies of exclusive graph-searching and min-CORDA.

Brief Announcement: Reaching Approximate Byzantine Consensus in Partially-Connected Mobile Networks*

Chuanyou Li[1], Michel Hurfin[2], and Yun Wang[1]

[1] School of Computer Science and Engineering, Southeast University, Nanjing, China
Key Lab of Computer Network & Information Integration, Ministry of Education
[2] INRIA Rennes Bretagne Atlantique, Campus de Beaulieu, Rennes, France
chuanyou.li@gmail.com, Michel.Hurfin@inria.fr, yunwang@seu.edu.cn

Abstract. We consider the problem of approximate consensus in mobile ad hoc networks in the presence of Byzantine nodes. Due to nodes' mobility, the topology is dynamic and unpredictable. We propose an approximate Byzantine consensus protocol which is based on the linear iteration method. In this protocol, nodes are allowed to collect information during several consecutive rounds: thus moving gives them the opportunity to gather progressively enough values. A novel sufficient and necessary condition guarantees the final convergence of the consensus protocol. At each stage of the computation, a single correct node is concerned by the requirement expressed by this new condition.

1 Context and Overview of the Proposed Protocol

We consider an ad hoc network composed of n nodes. When a node changes its physical location, it also changes the set of its neighbors. The system is unreliable. At most f nodes may suffer from Byzantine faults and messages may be lost. The parameter f is known by all correct nodes while the value of n is unknown. Among the variants of the consensus problem, one is called the *Approximate consensus* problem and has been presented for the first time in [1]. Each node begins to participate by providing a real value called its initial value. Eventually all correct nodes must obtain final values that are different from each other within a maximum value denoted ϵ (convergence property) and must be in the range of initial values proposed by the correct nodes (validity property).

To solve this problem in the presence of Byzantine nodes, some protocols [1] assume that the network is fully connected. Other protocols [2] consider partially connected networks but require an additional constraint: any correct node must know the whole topology. Based on the linear iterative consensus strategy, recent protocols [3,4] also assume that the network is partially connected but do not require any global information. At each iteration, a correct node broadcasts its

* This work is partially supported by Natural Science Foundation, China under grant 60973122, by National 863 Hi-Tech Program, China under grant 2011AA040502 and by the ANR INS French program (grant #ANR-11-INSE-010, project AMORES).

value, gathers values from its neighborhood and updates its own value. Its new value is an average of its own previous value and those of some of its neighbors. Like in [1], before computing its new value, a correct node must ignore some of the values it has collected (usually the f smallest and the f largest ones). These removed values may have been proposed by Byzantine nodes and may invalidate the validity property. In order to achieve convergence, the proposed solutions rely on additional conditions that have to be satisfied by the topology. In [3,4], the proposed conditions are proved to be sufficient and necessary in the case of an arbitrary directed graph. Yet in all these works, the condition refers only to the topology and is "universal": each node must always have enough neighbors (robustness of the network topology) and so none can be temporarily isolated.

Our protocol (described in [5]) is designed to cope with mobility. Each node follows an iteration scheme and repeatedly executes rounds. During a round, a node moves to a new location, broadcasts its current value, gathers values from its neighbors, and possibly updates its value. Our protocol differs from previous works for two main reasons. First, at the end of a round, the values used by a node to compute its new value are not only those that have been received during the current round. A node can take into account values contained in messages sent during some previous rounds. An integer parameter R_c is used to define the maximal number of rounds during which values can be gathered and stored while waiting to be used. Thanks to this flexibility, a node can use its ability to travel to collect step by step enough values. The second main difference is related to the sufficient and necessary condition used by our protocol. The proposed condition focuses on the dynamic subset of nodes that have currently either the minimal or the maximal value. Every R_c rounds at least one of these nodes must receive enough messages (quantity requirement) with values different from its current value (quality requirement). The condition is not universal (a single node is concerned) and considers both the topology and the values proposed by correct nodes. If $n \geq 3f + 1$, the condition can be satisfied. We are working on mobility scenarios (random trajectories, predefined trajectories, meeting points) to assert that the condition can be satisfied for reasonable values of R_c.

References

1. Dolev, D., Lynch, A.N., Pinter, S., Stark, W.E., Weihl, E.W.: Reaching approximate agreement in the presence of faults. In: Proc. of 3rd IEEE Symp. on Reliability in Distributed Software and Database Systems, pp. 145–154 (1983)
2. Sundaram, S., Hadjicostis, C.N.: Distributed function calculation via linear iterations in presence of malicious agents - part i: Attacking the networks. In: Proc. of the American Control Conference, pp. 1350–1355 (2008)
3. Vaidya, N., Tseng, L., Liang, G.: Iterative approximate byzantine consensus in arbitrary directed graphs. In: Proc. of 31st Symp. on PODC (2012)
4. Le Blanc, H., Zhang, H., Sundaram, S., Koutsoukos, X.: Consensus of multi-agent networks in the presence of adversaries using only local information. In: Proc. of the 1st Int. Conf. on High Confidence Networked Systems, pp. 1–10 (2012)
5. Li, C., Hurfin, M., Wang, Y.: Reaching approximate byzantine consensus in partially-connected mobile networks. Technical Report 7985, INRIA (May 2012)

Brief Announcement: Distributed Algorithms for Maximum Link Scheduling in the Physical Interference Model

Guanhong Pei[1] and Anil Kumar S. Vullikanti[2]

[1] Dept. of Electrical and Computer Engineering, and Virginia Bioinformatics Institute, Virginia Tech, USA
somehi@vt.edu

[2] Dept. of Computer Science, and Virginia Bioinformatics Institute, Virginia Tech, USA
akumar@vbi.vt.edu

Abstract. We develop distributed algorithms for the maximum independent link set problem in wireless networks in a distributed computing model based on the physical interference model with SINR constraints — this is more realistic and more challenging than the traditional graph-based models. Our results give the first distributed algorithm for this problem with polylogarithmic running time with a constant factor approximation guarantee, matching the sequential bound.

The Physical Interference Model. Graph-based interference models are widely used in wireless networking due to their simplicity, but have several limitations. A more realistic model is the physical interference model is based on SINR (Signal to Interference and Noise Ratio) constraints, referred to as the SINR model: a subset $L' \subseteq L$ of links can make successful transmission simultaneously if and only if the following condition holds for each $l \in L'$: $\frac{P(l)/d^{\alpha}(l)}{\sum_{l' \in L' \setminus \{l\}} P(l')/d^{\alpha}(l',l)+N} \geq \beta$, where $\alpha > 2$ is the "path-loss exponent", $\beta > 1$ is the minimum SINR required, N is the background noise, and $\phi > 0$ is a constant (note that α, β, ϕ and N are all constants). Here, $P(l)$ denotes the transmission power and $d(l)$ denotes the length of l, and $d(l', l)$ denotes the distance between the sender of l' and the receiver of l.

The Maximum Link Scheduling Problem (MAXLSP). Given a set L of communication requests on wireless links, the goal is to find a maximum independent subset of links that can transmit successfully at the same time in the SINR model. We study MAXLSP$^{\mathrm{U}}$, an instance of MAXLSP with uniform power level for data transmission.

Problem Background. The physical interference model is deemed as more realistic and much more challenging than graph-based ones for research in wireless networks. Under this model, MAXLSP$^{\mathrm{U}}$ is NP-Complete [1]; constant-approximation centralized algorithms have been proposed [2,3]. The only known distributed algorithm achieving constant approximation is [4] with long running time of $\Theta(m \log m)$ (where $m = |L|$). The difficulties of algorithm design lies in the following aspects: (1) the sender-receiver separation in the distributed setting, because the interference affects the reception on the receivers while it is the senders that decide whether to be included in S; (2) while we want to remove all the links that may either cause "large" interference on selected links or be "largely" interfered by the selected links, we also need to ensure this is done in an accurate fashion, so that the selected set of links is not over-trimmed; and (3) the interference is non-local, non-linear and cumulative.

Link Diversity and Link Classes. We define *link diversity* $g(L) \triangleq \lceil \log_2 \frac{d_{max}}{d_{min}} \rceil$, where d_{max} and d_{min} denote the maximum and minimum link lengths; in practical instances, $g(L)$ is a constant. Partition L into a set $\{L_i\}$ $(i = 1, 2, \ldots, g(L))$ of *link classes*, so that $L_i = \{l \mid 2^{i-1}d_{min} \leq d(l) < 2^i d_{min}\}$ is the set of links of roughly similar lengths.

SINR-Based Distributed Computing Model. The distributed model in the context of physical interference has not been studied extensively. We clearly summarize the main aspects as follows: (1) The network is synchronized and has time slots of unit length; (2) Nodes have a common estimate of m, within a polynomial factor; (3) Nodes share a common estimate of d_{min} and d_{max}; (4) We use only RSSI measurement from carrier sensing in our algorithm as a way of communication such that nodes gain information by only "listening" the channel rather than understanding messages. Thus, in our algorithm nodes transmit only "dumb" signals, which simplifies and speeds-up the process.

(ω_1, ω_2)-Ruling. Let W, W' denote two node sets. We say a node u is *ω-covered* by W', if and only if $\exists u' \in W', d(u, u') \leq \omega$. An (ω_1, ω_2)-*ruling* (where $\omega_1 < \omega_2$) of W is a node set denoted by $R_{\omega_1,\omega_2}(W)$, such that (1) $R_{\omega_1,\omega_2}(W) \subseteq W$; (2) all the nodes in $R_{\omega_1,\omega_2}(W)$ are at least ω_1-separated; that is, $\forall u, u' \in R_{\omega_1,\omega_2}(W)$, $d(u, u') \geq \omega_1$; and (3) W is ω_2-covered by $R_{\omega_1,\omega_2}(W)$.

Distributed Algorithm for MAXLSP$^{\mathrm{U}}$. We design an algorithm sweeping through all link classes in $g(L)$ phases, each of which consists of two steps. In the ith phase, at the first step, all previously selected links transmit and let the senders of the links in L_i sense and estimate the interference from all smaller link classes. If the interference exceeds a threshold on a link, the link quits. For the second step, we design a randomized distributed algorithm to find an (ω_1, ω_2)-ruling R_i of the set of senders of the remaining links of L_i in $\Theta(\log^3 m)$ time and select all links with their senders in R_i (where $m = |L|$); we embed in the ruling construction the removal of nearby links from $\cup_{j \geq i} L_j$ to control interference from all remaining link classes. The details and analysis can be found in [5]. Theorem 1 summarizes the performance of our algorithm.

Theorem 1. *There exists a distributed algorithm that computes an $O(1)$-approximation for* MAXLSP$^{\mathrm{U}}$ *in $O(g(L) \log^3 m)$ time in the pysical interference model,* w.h.p.

References

1. Goussevskaia, O., Oswald, Y.A., Wattenhofer, R.: Complexity in geometric sinr. In: ACM MobiHoc (2007)
2. Goussevskaia, O., Halldórsson, M., Wattenhofer, R., Welzl, E.: Capacity of arbitrary wireless networks. In: IEEE INFOCOM (2009)
3. Wan, P.-J., Jia, X., Yao, F.: Maximum Independent Set of Links under Physical Interference Model. In: Liu, B., Bestavros, A., Du, D.-Z., Wang, J. (eds.) WASA 2009. LNCS, vol. 5682, pp. 169–178. Springer, Heidelberg (2009)
4. Ásgeirsson, E., Mitra, P.: On a game theoretic approach to capacity maximization in wireless networks. In: IEEE INFOCOM (2011)
5. Pei, G., Kumar, V.S.A.: Distributed algorithms for maximum link scheduling under the physical interference model (2012), `http://arxiv.org/abs/1208.0811`

Brief Announcement: A Fast Distributed Approximation Algorithm for Minimum Spanning Trees in the SINR Model

Maleq Khan[1], Gopal Pandurangan[3], Guanhong Pei[1], and Anil Kumar S. Vullikanti[1,2]

[1] Virginia Bioinformatics Institute, Virginia Tech, USA
{maleq,somehi,akumar}@vbi.vt.edu
[2] Dept. of Computer Science, Virginia Tech, USA
[3] Division of Mathematical Sciences, Nanyang Technological University, Singapore, and Department of Computer Science, Brown University, USA
gopalpandurangan@gmail.com

Abstract. We study the *minimum spanning tree* (MST) construction problem in wireless networks under the physical interference model based on SINR constraints. We develop the first distributed (randomized) $O(\mu)$-approximation algorithm for MST, with the running time of $O(D \log n)$ (with high probability) where D denotes the diameter of the disk graph obtained by using the maximum possible transmission range, and $\mu = \log \frac{d_{max}}{d_{min}}$ denotes the "distance diversity" w.r.t. the largest and smallest distances between two nodes. (When $\frac{d_{max}}{d_{min}}$ is n-polynomial, $\mu = O(\log n)$.)

The Physical Interference Model. In recent years, the physical interference model based on SINR (Signal to Interference and Noise Ratio) constraints, referred to as the SINR model, has been found to be a more realistic model of wireless interference. In this model, a subset $L' \subseteq L$ of links can make successful transmission simultaneously if and only if the following condition holds for each $l \in L'$: $\frac{P(l)/d^{\alpha}(l)}{\sum_{l' \in L' \setminus \{l\}} P(l')/d^{\alpha}(l',l)+N} \geq \beta$, where $\alpha > 2$ is the "path-loss exponent", $\beta > 1$ is the minimum SINR required, N is the background noise, and $\phi > 0$ is a constant (note that α, β, ϕ and N are all constants). Here, $P(l)$ denotes the transmission power and $d(l)$ denotes the length of l, and $d(l', l)$ denotes the distance between the sender of l' and the receiver of l.

MST-SINR: The Minimum Spanning Tree Problem under the SINR Model. Given a set V of wireless nodes with a sink node s, the goal is to find a spanning tree T which minimizes $cost(T) = \sum_{(u,v) \in T} d(u, v)$, in a distributed manner. In particular, we are interested in ensuring that the communication at each step can be implemented in the SINR model. The spanning tree needs to be constructed implicitly, so that each node $u \in V$ only needs to find and know the id of its parent node $par(u)$.

Problem Background. The physical interference model is deemed as more realistic and much more challenging than graph-based ones for research in wireless networks. In recent years, distributed algorithms have been developed for a few fundamental "local" problems under this model, such as for maximum independent set, coloring, maximum dominating set problems. However, classical "global" problems such as minimum spanning tree, shortest path problems require an algorithm to "traverse" the entire network, and therefore more challenging for distributed solutions. Such problems remain open in a distributed setting, where the network diameter D is an inherent lower bound.

SINR-Based Distributed Computing Model. Let $r_{max} = (P_{max}/c)^{1/\alpha}$ denote the maximum transmission range of any node at the maximum power level. We summarize the main aspects of our distributed model in the context of physical interference as follows: (1) The network is synchronized with unit slots. (2) The network is connected w.r.t. a range r_{max}/c for some constant c; (3) Nodes have a common estimate of n, within a polynomial factor. (4) Nodes share a common estimate of $d_{min} = 1$ and d_{max}, the minimum and maximum distances between nodes. (5) We assume nodes are equipped with software-defined radios and can transmit at any power level $P \in [1, P_{max}]$. (6) The success of communication is determined by SINR.

Distributed Algorithm for MST. Our distributed algorithm is based on the Nearest Neighbor Tree (NNT) scheme [1] which consists of two steps: (1) each node first chooses a unique *rank*; (2) each node connects to the *nearest* node of higher rank. Our algorithm involves two stages: "bottom-up" and "top-down."

(1) During the bottom-up stage of $\Theta(\mu \log n)$ slots, for some range $r'_{max} < r_{max}$, we run $\log r'_{max} \leq \mu$ phases, ranging from $i = 1, \ldots, \log r'_{max}$. In the ith phase,
   (i) a subset S_i of nodes participate, and the edges chosen so far form a forest rooted at nodes in S_i. The nodes in S_i transmit at power level of $c \cdot d_i^{\alpha}$ for a constant c, where $d_i = 2^i$.
   (ii) each node $v \in S_i$ approximates the NNT scheme by connecting to a "close-by" node in S_i within distance $c' \cdot d_i$ of higher rank, if there exists one, for a constant c'. The nodes which are not able to connect continue to phase $i + 1$.

   At the end of the bottom-up stage, we obtain a forest.

(2) During the top-down stage of $\Theta(D \log n)$ slots, we first form a constant-density dominating set Dom with some range so that
   (i) each node $v \notin Dom$ is within distance r'_{max} of some node in Dom, and
   (ii) for each node $u \in Dom$, the number of nodes within range r'_{max} is "small".

   We then assign ranks from the sink to the periphery of the network using local broadcast at each step (taking advantage of the constant density). This leads to connecting all the forests produced in the first stage.

The details and analysis can be found in the full paper [2]. Theorem 1 summarizes the performance of our algorithm. Our algorithm's running time is essentially optimal (upto a logarithmic factor), since computing *any* spanning tree takes $\Omega(D)$ time.

Theorem 1. *There exists a distributed algorithm that produces a spanning tree of cost $O(\mu)$ times the optimal in time $\Theta(D \log n)$,* w.h.p.*, in the SINR model.*

References

1. Khan, M., Pandurangan, G., Kumar, V.S.A.: Distributed algorithms for constructing approximate minimum spanning trees in wireless sensor networks. IEEE Transactions on Parallel and Distributed Systems 20(1), 124–139 (2009)
2. Khan, M., Kumar, V.S.A., Pandurangan, G., Pei, G.: A fast distributed approximation algorithm for minimum spanning trees in the sinr model (2012), `http://arxiv.org/abs/1206.1113`

Brief Announcement: Deterministic Protocol for the Membership Problem in Beeping Channels

Bojun Huang

Microsoft Research Asia
bojhuang@microsoft.com

The *beeping channel* model is a multiple access channel (MAC) model where active nodes can only send/hear a "jamming" signal (i.e. a beep) through the communication channel in each time slot [2]. A listening node hears a beep signal if at least one node is beeping; otherwise it hears nothing. The beeping model was recently proposed to model carrier-sensing-based wireless communication [2], and the Delta-Notch signalling mechanism between biological cells [1]. The motivation of our work, however, is to design efficient digital circuits. It turns out that the beeping channel model well characterizes the behaviors of a group of *sequential logic* modules connected by a logical-OR gate. A strictly synchronized global clock is available in such a circuit. In a clock cycle, a high electrical level in each input wire of the OR gate corresponds to the choice to beep made by the module connecting to this input wire while a low level corresponds to the choice not to beep. The output of the OR gate is wired back to each module as the source signal in the next cycle. We focus on deterministic protocols, which is preferred in hardware design and other applications requiring safety guarantee.

The membership problem is a re-formuation of the classic *conflict resolution* problem under the beeping channel model. Suppose n nodes connect to a beeping channel. An *active* node can choose to either listen or beep in each time slot, while an inactive node can only listen to the channel. The activeness of every node does not change over time. Each node is assumed to know its identifier $i \in \{1, ..., n\}$, its activeness status, and the values of k and n. A (n, k)-*membership* problem asks to let the n nodes (including the inactive ones) agree on the identifiers of the at most k active nodes, using as few time slots as possible. Sometimes, parallel access to multiple independent channels is possible, such as in 64-bit circuits. In this case, the performance of a membership protocol is measured by the number of *stages*, where a stage is defined as a batch of consecutive time slots that can run in parallel. The number of stages corresponds to the number of time slots when an unlimited number of channels are available.

The membership problem can be solved by reducing to *group testing* [3], a famous combinatorial searching problem that asks to identify the set of *positive* individuals from a large population, using as few *tests* as possible. A test is a query to a subset of the individuals, which returns positive if there is at least one in the subset queried. One test in group testing corresponds to the running of the beeping channel in one time slot. The main difference between the two problems is that a node in the beeping channels has the additional information of its local activeness status, while the tester in group testing knows nothing about the population to be tested. Group testing is known to have a $\Omega(k \log \frac{n}{k})$ lower bound in tests, and several *adaptive* algorithms have matched this lower bound [3]. However, when equivalently rewriting these adaptive group testing algorithms into membership protocols, the resulting protocols will also require $\Omega(k \log \frac{n}{k})$ stages, which is not efficient in the multi-channel scenario. Therefore, it remains open to design a practical deterministic membership protocol that is efficient both in time and in stage.

Our Contributions. We first rigorously prove the equivalence between the (n, k)-membership problem in beeping channels and group testing, which leads to a tight lower bound of $\Omega(k \log \frac{n}{k})$ time slots for the former. Especially, we prove that it's impossible to utilize the activeness information of each node for any deterministic membership protocol in the beeping channels.

Another main result of this work is to propose and analyze a practical membership protocol, called *the funnel protocol*, which is efficient both in time and in stage. The basic idea is to iteratively reduce the problem size n in each stage by renaming the active nodes based on a smaller name space. Specifically, in each stage, an active node encodes its current identifier into a binary string and beeps out the codeword (it takes l time slots to beep a l-bit codeword). A candidate set of the "possibly" active nodes can be determined by analyzing the channel feedback, and then each active node renames itself according to the order of its current identifier in this candidate set. All inactive nodes are filtered out when the protocol terminates, and each node can locally recover the original identifiers of all the active nodes from the channel feedback history. We use the standard *identity code* to encode identifiers in the funnel protocol. The iteration reduces a (n, k)-membership problem to a (k^d, k)-membership problem within $dn^{\frac{1}{d}} + d$ time slots and 1 stage, where $d \in \mathbb{Z}^+$ is a controlling factor and always equals 1 in the last stage. The choices for d at each and every stage form an iteration schedule, which finally determines the performance of the funnel protocol. By analytically solving the discrete Bellman equation defined in Eq. (1), we show that the optimal iteration schedule of the funnel protocol uses $O(k \log \frac{n}{k} + k^2)$ time slots and $O(\log k \log \log n)$ stages, which requires exponentially fewer stages than existing adaptive protocols. The funnel protocol can be further combined with an adaptive protocol to form a hybrid protocol with $O(k \log \frac{n}{k})$ time slots and $O(\log k \log \log n + k(\log k)^2)$ stages. All constant factors hidden in the big-O's above are small. Moreover, the funnel protocol can always identify all the nodes remaining active when it terminates even if the nodes crash arbitrarily.

$$f_k(n) = \min_{1 \le d \le log_k n} \{dn^{\frac{1}{d}} + d + f_k(k^d)\} \qquad n, k, d \in \mathbb{Z}^+, \quad f_k(k) = 0 \qquad (1)$$

References

1. Afek, Y., Alon, N., Barad, O., Hornstein, E., Barkai, N., Bar-Joseph, Z.: A Biological Solution to a Fundamental Distributed Computing Problem. Science 331(6014), 183–185 (2011)
2. Cornejo, A., Kuhn, F.: Deploying Wireless Networks with Beeps. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 148–162. Springer, Heidelberg (2010)
3. Du, D., Hwang, F.: Combinatorial Group Testing and Its Applications (2000)

Brief Announcement: Probabilistic Stabilization under Probabilistic Schedulers[⋆]

Yukiko Yamauchi[1], Sébastien Tixeuil[2], Shuji Kijima[1], and Masafumi Yamashita[1]

[1] Kyushu University, Japan
{yamauchi,kijima,mak}@inf.kyushu-u.ac.jp
[2] UPMC Sorbonne Universites, France
Sebastien.Tixeuil@lip6.fr

Motivation. Roughly speaking, a weakly stabilizing system $\mathcal{S}$ executed under a probabilistic scheduler ρ is probabilistically self-stabilizing, in the sense that any execution eventually reaches a legitimate execution with probability 1 [1–3]. Here ρ is a set of Markov chains, one of which is selected for $\mathcal{S}$ by an *adversary* to generate as its evolution an infinite activation sequence to execute $\mathcal{S}$. The performance measure is the worst case expected convergence time $\tau_{\mathcal{S},M}$ when $\mathcal{S}$ is executed under a Markov chain $M \in \rho$. Let $\tau_{\mathcal{S},\rho} = \sup_{M \in \rho} \tau_{\mathcal{S},M}$. Then $\mathcal{S}$ can be "comfortably" used as a probabilistically self-stabilizing system under ρ only if $\tau_{\mathcal{S},\rho} < \infty$. There are $\mathcal{S}$ and ρ such that $\tau_{\mathcal{S},\rho} = \infty$, despite that $\tau_{\mathcal{S},M} < \infty$ for any $M \in \rho$. Somewhat interesting is that, for some $\mathcal{S}$, there is a randomised version $\mathcal{S}^*$ of $\mathcal{S}$ such that $\tau_{\mathcal{S}^*,\rho} < \infty$, despite that $\tau_{\mathcal{S},\rho} = \infty$, i.e., randomization helps. This motivates a characterization of $\mathcal{S}$ that satisfies $\tau_{\mathcal{S}^*,\rho} < \infty$.

Model. A *distributed system* is defined by a pair $\mathcal{S} = (N, \mathcal{A})$ of a communication graph N and a distributed algorithm $\mathcal{A}$. A communication graph $N = (P, L)$ is a directed graph, where P is the set of processes and L is the set of communication links. Here $(p, q) \in L$ means that the local variables of p are visible from q. $\mathcal{A} = \{A_p : p \in P\}$ is a set of local algorithms A_p for $p \in P$. A_p specifies the set of local variables and their ranges, and hence defines the set of local configurations $\mathit{\Gamma}_p$. Then $\mathit{\Gamma} = \mathit{\Pi}_{p \in P} \mathit{\Gamma}_p$ is the set of all global configurations. If $\mathcal{S}$ is in $\gamma \in \mathit{\Gamma}$ and the processes in $W \subseteq P$ are activated, then in each $p \in W$, A_p is invoked to update its local variables (based on the values of variables visible from p, including those in p), to yield a configuration $\delta(\gamma, W) \in \mathit{\Gamma}$. Without loss of generality, we may assume that A_p updates at least one variable, so that $\delta(\gamma, W) \neq \delta(\gamma, W')$ if and only if $W \neq W'$. We identify $\mathcal{S}$ with a directed graph $(\mathit{\Gamma}, T)$ with edge labels W, where $T = \{(\gamma, \delta(\gamma, W), W) : \gamma \in \mathit{\Gamma}, W \subseteq P\}$. By definition two edges leaving from the same configuration have different labels.

A *probabilistic scheduler* ρ_F is the set of all finite state Markov chains M augmented by edge labels from 2^P in such a way that two edges (i.e., two transitions with positive transition probabilities) leaving from the same state have different labels. An *execution* of $\mathcal{S}$ under ρ_F is a sequence $\mathcal{E} = \gamma_0, \gamma_1, \ldots$ of random variables γ_t representing the configuration at time t for any $t \geq 0$, and is defined

[⋆] This work is supported in part by MEXT/IPSJ KAKENHI (21650002, 22300004, 23700019, 24104003, and 24650008), ANR project SHAMAN and JSPS fellowship.

as follows: An adversary first selects a Markov chain $M \in \rho_F$. Let $Z_t \subseteq P$ be the random variable to represent the label attached to the t-th transition of M. Let $W_t = Z_t \cap Y_t$, where Y_t represents the set of enabled processes at γ_t. Then γ_{t+1} is the unique configuration such that $(\gamma_t, \gamma_{t+1}, W_t) \in T$. We also denote the conventional (deterministic strongly) fair scheduler by σ_F.

This paper considers only a simple randomization of $\mathcal{A}$. In a randomised version $\mathcal{A}^*$, whenever $\mathcal{A}_p$ is invoked, p renounces the privilege to execute it with some prespecified probability, which may depend on p and the current values of variables visible from p. Letting this probability $\mathcal{D}$, we denote $\mathcal{A}^*$ by $\langle \mathcal{A}, \mathcal{D} \rangle$, and the corresponding randomized system $\mathcal{S}^*$ by $\langle \mathcal{S}, \mathcal{D} \rangle$. Note that if $\mathcal{D} = 0$ then $\mathcal{A}^* = \mathcal{A}$ and $\mathcal{S}^* = \mathcal{S}$, and that if $\mathcal{D} = 1$ then no progress is made.

A *specification* $\mathcal{SP}$ for $\mathcal{S}$ is the set of correct executions. A configuration $\gamma \in \Gamma$ is *legitimate* for $\mathcal{SP}$, if any execution starting from γ is in $\mathcal{SP}$. $\mathcal{S}$ is *weakly stabilizing* for $\mathcal{SP}$ under σ_F, if any $\gamma \in \Gamma$ has at least one execution starting from γ under σ_F that reaches a legitimate configuration. $\langle \mathcal{S}, \mathcal{D} \rangle$ is *probabilistically stabilizing* for $\mathcal{SP}$ under ρ_F if any execution under ρ_F reaches a legitimate configuration with probability 1.

Theorem 1. *$\mathcal{S}$ is weakly stabilizing for $\mathcal{SP}$ under σ_F, if and only if $\langle \mathcal{S}, \mathcal{D} \rangle$ is probabilistically self-stabilizing for $\mathcal{SP}$ under ρ_F provided* $0 < \mathcal{D} < 1$.

Indeed, we can show that $\tau_{\mathcal{S}^*,M} < \infty$ for any $M \in \rho_F$ if $\mathcal{S}$ is weakly stabilizing for $\mathcal{SP}$ under σ_F. However it does not necessarily mean $\tau_{\mathcal{S}^*,\rho_F} < \infty$. Given $\mathcal{S}$ and $\mathcal{SP}$, we contract all the legitimate states of $\mathcal{S}$ into a newly introduced state γ_L and let $\hat{\mathcal{S}}$ be the result. Next let $\hat{\mathcal{S}}_p$ be the graph constructed from $\hat{\mathcal{S}}$ by removing all edges whose labels are *not* $p \in P$. $\mathcal{S}$ is said to be *regular* for $\mathcal{SP}$ if $\hat{\mathcal{S}}_p$ is a spanning in-tree rooted γ_L for each $p \in P$.

Theorem 2. *$\mathcal{S}$ is regular for $\mathcal{SP}$, if and only if $\tau_{\mathcal{S}^*,\rho_F} < \infty$.*

Note that the same property holds even for some restricted classes of probabilistic schedulers, such as the probabilistic central scheduler (which activates only a singleton), and the probabilistic memory-less scheduler (which consists of all single state Markov chains).

Finally, we implicitly assumed $|\Gamma| < \infty$ in above. When $|\Gamma| = \infty$, we have the following theorem. Let h_p be the height of $\hat{\mathcal{S}}_p$ when $\mathcal{S}$ is regular.

Theorem 3. *Suppose that $|\Gamma| = \infty$. $\mathcal{S}$ is regular for $\mathcal{SP}$ and $h_p < \infty$ for all $p \in P$, if and only if $\tau_{\mathcal{S}^*,\rho_F} < \infty$.*

References

1. Devismes, S., Tixeuil, S., Yamashita, M.: Weak vs. self vs. probabilistic stabilization. In: Proc. of ICDCS 2008, pp. 681–688 (2008)
2. Gouda, M.G.: The Theory of Weak Stabilization. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 114–123. Springer, Heidelberg (2001)
3. Herman, T.: Probabilistic self-stabilization. IPL 35(2), 63–67 (1990)

Brief Announcement: 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

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 [4,5], distributed queues [2], and mobile object tracking [1]. 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 getting 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, e.g. [2], 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.

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 (e.g., Arrow [3]), 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 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 over the hierarchy to the optimal communication cost of serving them over the original network. In the analysis, we focus only the *move* requests since they are the most costly operations. Further, we consider only one shared object as in [3].

A node u in each level k has a *write set* of leaders which helps to implement the *move* requests. Let η be a write size related parameter which expresses what is the maximum size of the write set of leaders of the node u among all the levels in the hierarchy, φ be a stretch related parameter which expresses how far the leaders in the write set of u can appear beyond a minimum radius around u, and σ be a growth related parameter which expresses the minimum radius growth ratio on the hierarchy. We prove:

Theorem 1. *Any distributed hierarchical directory is $\mathcal{O}(\eta \cdot \varphi \cdot \sigma^3 \cdot h)$-competitive for any arbitrary set of (online)* move *requests in dynamic executions.*

We apply our framework to analyze three variants of distributed hierarchical directory-based protocols, Spiral [5], Ballistic [4], and Awerbuch and Peleg's tracking a mobile user [1] (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 [4].
- 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.
- Ballistic: $\mathcal{O}(\log D)$ competitive ratio in constant-doubling networks. It is also for the *data-flow* distributed implementation of software transactional memory.

These bounds subsume the previous bounds for these protocols [1,4,5] on both *sequential executions* which consist of non-overlapping sequence of requests and *one-shot concurrent executions* where all requests appear simultaneously.

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 also for any asynchronous execution of the requests. We perform the analysis level by level. The main idea is to analyze separately the windows which contain many requests, the *dense* windows, and the windows which contain few requests, the *sparse* windows. In summary, 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.

References

1. Awerbuch, B., Peleg, D.: Concurrent online tracking of mobile users. SIGCOMM Comput. Commun. Rev. 21(4), 221–233 (1991)
2. 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)
3. Herlihy, M., Kuhn, F., Tirthapura, S., Wattenhofer, R.: Dynamic analysis of the arrow distributed protocol. Theor. Comp. Sys. 39(6), 875–901 (2006)
4. Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distributed Computing 20(3), 195–208 (2007)
5. Sharma, G., Busch, C., Srinivasagopalan, S.: Distributed transactional memory for general networks. In: IPDPS, pp. 1045–1056 (2012)

Brief Announcement: Flooding in Dynamic Graphs with Arbitrary Degree Sequence*

Hervé Baumann[1], Pierluigi Crescenzi[2], and Pierre Fraigniaud[1]

[1] CNRS and Univ. Paris Diderot
[2] Università degli Studi di Firenze, Italy

1. Introduction. The simplest communication mechanism that implements the broadcast operation is the *flooding* protocol, according to which the source node is initially informed, and, when a not informed node has an informed neighbor, then it becomes informed at the next time step. In this paper we study the flooding *completion time* in the case of dynamic graphs with arbitrary degree sequence, which are a special case of random evolving graphs. A *random evolving graph* is a sequence of graphs $(G_t)_{t\geq 0}$ with the same set of nodes, in which, at each time step t, the graph G_t is chosen randomly according to a probability distribution over a specified family of graphs. A special case of random evolving graph is the *edge-Markovian* model (see the definition below), for which tight upper bounds on the flooding completion time have been obtained by using a so-called *reduction lemma*, which intuitively shows that the flooding completion time of an edge-Markovian evolving graph is equal to the diameter of a suitably defined weighted random graph. In this paper, we show that this technique can be applied to the analysis of the flooding completion time in the case of a random evolving graph based on the following generative model. Given a sequence $\mathbf{w} = w_1, \ldots, w_n$ of non-negative numbers, the graph $G_{\mathbf{w}}$ is a random graph with n nodes in which each edge (i, j) exists with probability $p_{i,j} = \frac{w_i w_j}{\sum_{k=1}^{n} w_k}$ (independently of the other edges). It is easy to see that the expected degree of node i is w_i: hence, if we choose $\mathbf{w}$ to be a sequence satisfying a power law, then $G_{\mathbf{w}}$ is a power-law graph, while if we choose $w_i = pn$, then $G_{\mathbf{w}}$ is the $G_{n,p}$ Erdös-Rényi random graph.

2. Our Results. Let $f(\mathcal{S}, i)$ denote the number of steps it takes for the flooding protocol to propagate a message initiated at node i in a sequence $\mathcal{S}$ of random graphs picked in $\mathcal{G}_{\mathbf{w}}$, and let $f(\mathcal{S}) = \max_{i\in[n]} f(\mathcal{S}, i)$. We define the randomly-weighted graph H, called the *weighted representative graph* of $\mathcal{G}_{\mathbf{w}}$, as follows. H is the n-node clique whose edges have weights. The weight $\mathit{weight}(e_{i,j}) \geq 1$ of edge $e_{i,j}$ between node i and node j is drawn at random according to the geometric distribution of parameter $p_{i,j}$ (that is, $\Pr\{\mathit{weight}(e_{i,j}) = k\} = p_{i,j}(1-p_{i,j})^{k-1}$). For $k \in \mathbb{N}^+ \cup \{\infty\}$, we define the (random) weighted graph $H^{(k)}$, obtained from H by removing all edges with weights greater than k (note that $H^{(1)}$ is distributed identically as any of the graphs in the sequence generated according to $\mathcal{G}_{\mathbf{w}}$, and that $H^{(\infty)} = H$). Let C be a connected component of $H^{(k)}$, and let $i \notin C$. We define $\mathit{connect}(i, C)$ as the random variable equal to the smallest h

* The first and third authors received additional supports from the ANR projects "DISPLEXITY" and "PROSE", and from the INRIA project "GANG". The second author was supported in part by funds of the "DISCO" National Research Project.

such that there is an edge from i to C in $H^{(h)}$, and we define $\mathit{connect}(C) = \max_{i \notin C} \mathit{connect}(i, C)$. Moreover, let $\mathit{diam}(C)$ be the weighted diameter of C, and let $\mathit{diam}_1(C)$ be the diameter of C when one ignores the weights.

Lemma 1. *For any $k \geq 1$, and any connected component C of $H^{(k)}$, $f(\mathcal{S}) \leq \mathit{diam}(C) + 2\,\mathit{connect}(C)$. Hence, $f(\mathcal{S}) \leq k\,\mathit{diam}_1(C) + 2\,\mathit{connect}(C)$.*

One can identify three scenarios of applications for Lemma 1, which provide a methodology for the analysis of flooding in dynamic networks.

- Scenario 1 $H^{(1)}$ is connected. Then, $f(\mathcal{S})$ is upper bounded by $\mathit{diam}(H^{(1)})$.
- Scenario 2 $H^{(1)}$ is not connected, but has a giant component C. Then, $f(\mathcal{S})$ is upper bounded by $\mathit{diam}(C)$ plus 2 $\mathit{connect}(C)$.
- Scenario 3 $H^{(1)}$ has no giant component. Then upper bounding the flooding time can be achieved by searching for the smallest value k such that $H^{(k)}$ has a giant component C. Once this is done, the diameter of C is upper bounded by k times the (unweighted) diameter of the giant component in an appropriately defined random graph model $\mathcal{G}'_{\mathbf{w}}$ related to k and $\mathcal{G}_{\mathbf{w}}$. Finally, we upper bound the flooding time by adding the bound on the diameter of C to the value of $\mathit{connect}(C)$ computed as in the second scenario.

By applying this methodology, we can prove that, in the case of a power-law degree sequence, the flooding completion time is almost surely logarithmic in n, and we can show several bounds on the flooding completion time, in the case of evolving graphs with an arbitrary given degree distribution $\mathbf{w}$.

3. Our Main Open Problem. As we already mentioned, Lemma 1 applies in the more general context of sequences of edge-Markovian graphs, where G_{t+1} is depending on G_t according to the following rule depending on two parameters, the birth rate $p_{i,j}$, and the death rate $q_{i,j}$, for every edge $e_{i,j}$, $1 \leq i, j \leq n$. Every edge $e_{i,j}$ not present in G_t appears in G_{t+1} with probability $p_{i,j}$, while every edge $e_{i,j}$ present in G_t disappears in G_{t+1} with probability $q_{i,j}$, in a way mutually independent from the behavior of all the other edges. For instance, in the case $p_{i,j} = p$ and $q_{i,j} = q$ for all i, j, the steady state of this Markovian process is a random graph in $\mathcal{G}_{n,\widehat{p}}$ where $\widehat{p} = p/(p+q)$. Our results raise the issue of designing a "natural" Markovian process guiding the appearance and disappearance of edges so that to produce a sequence of graphs whose steady state is $\mathcal{G}_{\mathbf{w}}$, for every given $\mathbf{w}$. Of course, setting $p_{i,j} = \frac{w_i w_j}{\sum_{k=1}^{n} w_k}$, and setting $q_{i,j} = 1 - p_{i,j}$ provides such a sequence, but it does not include time-dependencies, for the graph G_{t+1} is actually independent of G_t. In "practice", dynamic networks enjoy time-dependencies, and thus the design of a tractable and realistic model for a Markovian sequence of graphs with, e.g., power-law distribution is highly desirable. It is not clear whether the design of such a model is doable without introducing also some spatial-dependencies (i.e., dependencies between edges). The edge-Markovian model restricts the dynamic to be free of spatial dependencies. Whether this restriction prevents one from the ability of designing a model of dynamic graphs with given expected degree-distribution is an intriguing question.

Brief Announcement: Node Sampling Using Centrifugal Random Walks*

Andrés Sevilla[1], Alberto Mozo[2], and Antonio Fernández Anta[3]

[1] Dpto Informática Aplicada, U. Politécnica de Madrid, Madrid, Spain
[2] Dpto Arquitectura y Tecnología de Computadores, U. Politécnica de Madrid, Madrid, Spain
[3] Institute IMDEA Networks, Madrid, Spain
{asevilla,amozo}@eui.upm.es antonio.fernandez@imdea.org

Abstract. We propose distributed algorithms for sampling networks based on a new class of random walks that we call *Centrifugal Random Walks* (CRW). A CRW is a random walk that starts at a source and *always* moves *away* from it. We propose CRW algorithms for connected networks with arbitrary probability distributions, and for grids and networks with regular concentric connectivity with distance based distributions. All CRW sampling algorithms select a node with the exact probability distribution, do not need warm-up, and end in a number of hops bounded by the network diameter.

1 Introduction

Sampling the nodes of a network is the building block of epidemic information spreading [3], and can be used to construct small world network topologies [1]. A classical technique to implement distributed sampling is to use gossiping among network nodes [2]. A second popular distributed technique is the use of random walks [5]. Unfortunately, in these approaches, the desired probability distribution is reached when the stationary distribution of a Markov process is reached. The number of iterations (or hops of a random walk) required to reach this situation (the warm-up time) depends on the parameters of the network and the desired distribution, but it is not negligible.

We present efficient distributed algorithms to implement a sampling service. The basic technique used for sampling is a new class of random walks that we call *Centrifugal Random Walks* (CRW). A CRW starts at a network node, called the *source*, and *always* moves *away* from it. The sampling process in a *CRW algorithm* works essentially as follows. A CRW always starts at the source node. When the CRW reaches a node x (initially the source s), the CRW stops and selects that node with a given *stay probability*. If the CRW does not stop at x, it jumps to a neighbor of x that is farther away from the source than x. (The probability of jumping to each of these neighbors is not necessarily the same.)

Using this general approach, we firstly propose a CRW algorithm that samples *any* connected network with *any* probability distribution (given as nodes' weights). Before starting the sampling, a preprocessing phase is required. This preprocessing involves building a minimum distance spanning tree (MDST) in the network, and using this tree for efficiently aggregating the nodes' weights. Once the preprocessing is completed, any node in the network can be the source of a sampling process, and multiple independent

* Work partially supported by Comunidad de Madrid (S2009TIC-1692) and Spanish MICINN (TEC2011-29688-C02-01). Thanks to Husain Ibraheem for useful suggestions.

samplings with the exact desired distribution can be efficiently performed. Since the CRW used for sampling follow the MDST, they take at most D hops (where D is the network diameter).

Secondly, CRW algorithms without preprocessing are proposed when the probability distribution is distance-based (i.e., all the nodes at the same distance in hops from the source are selected with the same probability). The first distance-oriented CRW algorithm we propose samples with a distance-based distribution in a grid. In this network, the source node is at position $(0, 0)$ and the grid contains all the nodes that are at a distance no more than the radius R from the source. The algorithm we derive assigns a stay probability to each node that only depends on its distance from the source. However, the hop probabilities depend on the position (i, j) of a node and the position of the neighbor to which the CRW can jump. Since every jump of the CRW in the grid moves one hop away from the source, the sampling is completed after at most R hops.

For the general case of any connected network, we can picture nodes at each distance k from the source as positioned on a ring. The center of all the rings is the source, and the radius of each ring is one unit larger than the previous one. Using this graphical image, we refer the networks of this family as *concentric rings networks*. We have proposed a CRW algorithm that samples with distance-based distributions in concentric rings networks *with uniform connectivity*. These are networks in which all the nodes in each ring k have the same number of neighbors in ring $k - 1$ and the same number in ring $k + 1$. Like the grid algorithm, this one samples in at most R hops, where R is the number of rings. To deal with concentric rings networks with no uniform connectivity, we propose a distributed algorithm that, if it completes successfully, builds an overlay network that has uniform connectivity. In the resulting network, the algorithm for uniform connectivity can be used. We have found via simulations that this algorithm succeeds in building the desired overlay network in a significant number of cases.

In summary, CRW algorithms can be used to implement an efficient sampling service because, unlike previous Markovian, (e.g., random walks and gossiping) approaches, (1) they always finish in a number of hops bounded by the network diameter, (2) select a node with the *exact probability distribution*, and (3) do not need warm-up (stabilization) to converge to the desired distribution. Additionally, in the case that preprocessing is needed, this only has to be executed once, independently on the number of sources and the number of samples taken from the network. More details can be found at [4].

References

1. Bonnet, F., Kermarrec, A.-M., Raynal, M.: Small-World Networks: From Theoretical Bounds to Practical Systems. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 372–385. Springer, Heidelberg (2007)
2. Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.-M., van Steen, M.: Gossip-based peer sampling. ACM Trans. Comput. Syst. 25(3) (2007)
3. Kempe, D., Kleinberg, J.M., Demers, A.J.: Spatial gossip and resource location protocols. J. ACM 51(6), 943–967 (2004)
4. Sevilla, A., Mozo, A., Fernández Anta, A.: Node sampling using drifting random walks. CoRR, abs/1107.1089v2 (2012)
5. Zhong, M., Shen, K.: Random walk based node sampling in self-organizing networks. SIGOPS Oper. Syst. Rev. 40, 49–55 (2006)

Brief Announcement: Concurrent Wait-Free Red-Black Trees[⋆]

Aravind Natarajan, Lee Savoie, and Neeraj Mittal

The University of Texas at Dallas, Richardson TX 75080, USA
{aravindn@,lee.savoie@alumnimail,neerajm}@utdallas.edu

Motivation: With the prevalence of multi-core multi-processor systems, concurrent data structures are becoming increasingly important. Concurrency is most often managed through locks. However, lock-based implementations of concurrent data structures are vulnerable to problems such as deadlock, priority inversion and convoying. Non-blocking algorithms avoid the pitfalls of locks by using hardware-supported read-modify-write instructions such as load-linked/store-conditional (LL/SC) and compare-and-swap (CAS). In this announcement, we focus on a non-blocking concurrent red-black tree. Red-black tree is a type of self-balancing binary search tree that provides good worst-case time complexity for search and modify (insert, update and delete) operations. However, red-black trees have been remarkably resistant to parallelization using both lock-based and lock-free techniques. The tree structure causes the root and high level nodes to become the subject of high contention and thus become a bottleneck. This problem is only exacerbated by the introduction of balancing requirements. We present a suite of *wait-free* algorithms for concurrently accessing an external red-black tree, obtained through a progressive sequence of modifications to an existing general framework. In all our algorithms, search operations only execute read and write instructions on shared memory.

A Wait-Free Framework for Tree Based Data Structures: Tsay and Li described a framework in [1], henceforth referred to as the TL-framework, that can be used to develop wait-free operations for a tree-based data structure provided operations traverse (and modify) the tree in a top-down manner. The TL-framework is based on the concept of a *window*, which is simply a rooted subtree of the tree structure, that is, a small, contiguous piece of the tree. This window slides down the tree as the operation proceeds, and can dynamically change size. For an algorithm to fit into the TL-framework, it must operate on consecutive windows along a simple path from the root node toward a leaf node. An operation makes a local copy of all nodes within its current window and makes changes to the local copy. The current window is then replaced atomically by the local copy. If any node in the window is owned by another operation, that operation is moved out of the way by *recursive helping.* We refer to nodes reachable from the root of the tree as *active* nodes, while those no longer reachable as *passive* nodes.

In the TL-framework, every operation including search operation: (i) only "acts" on active nodes, (ii) needs to make a copy of every node that it encounters,

[⋆] This work was supported in part by the NSF Grant CNS-1115733.

and (iii) needs to help every stalled operation on its path before it can advance further. This copying and helping makes operations *expensive* to execute. We now summarize our contributions that aim at reducing the overhead of operations.

Reducing the Overhead of a Search Operation: Our first contribution is showing that search operations can simply traverse the tree, without copying nodes. Such *fast* search operations traverse the tree unaware of other operations, and without helping other operations complete. Note that a search operation can now access passive nodes, and make its decision based on "old" information. We prove that, even when this happens, the algorithm still generates linearizable executions.

Reducing the Overhead of a Modify Operation: The overhead of a modify operation can be reduced in practice by first using the fast search operation to determine whether the tree contains the key, and depending on the result, execute the modify operation using the TL-framework. In the case of an insert/update operation, if the search operation finds the key in the tree, then the value associated with the key is changed outside the TL-framework. Note that, in this case, the value has to be stored outside the node as a separate record, whose address is stored in the node. The value is then updated in a wait-free manner using the algorithm proposed by Chuong *et al.* [2]. We further reduce the overhead of a modify operation by copying nodes within a window only if the window is changed in some way. Note that in this case, a search operation may be *overtaken* by concurrent modify operations. To ensure wait-freedom, modify operations now need to help search operations complete.

A Customized Wait-Free Garbage Collector: As modify operations traverse the tree, they replace existing nodes with new copies. The replaced nodes are no longer accessible to any operation that starts thereafter. To ensure that the system does not run out of memory, we have designed a customized wait-free garbage collection scheme, that is based on the notion of *hazard pointers* [3]. Every process maintains its own hazard pointer list, that is manipulated using simple read and write instructions. Before reclaiming a node, a process scans the list of hazard pointers of each process to ensure that the address is not present. However, since search operations can access passive nodes in our algorithm, a process performing garbage collection must help all other concurrent search operations complete to ensure that they do not access a garbage collected node.

References

1. Tsay, J.J., Li, H.C.: Lock-Free Concurrent Tree Structures for Multiprocessor Systems. In: Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS), pp. 544–549 (December 1994)
2. Chuong, P., Ellen, F., Ramachandran, V.: A universal construction for wait-free transaction friendly data structures. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 335–344 (2010)
3. Michael, M.M.: Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Transactions on Parallel and Distributed Systems (TPDS) 15(6), 491–504 (2004)

Brief Announcement: A Contention-Friendly, Non-blocking Skip List[*]

Tyler Crain[1], Vincent Gramoli[2], and Michel Raynal[1,3]

[1] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
[2] The University of Sydney, NSW 2006, Australia
[3] Institut Universitaire de France

A skip list is a probabilistic data structure to store and retrieve in-memory data in an efficient way. In short, it is a linked structure that diminishes the linear big-oh complexity of a linked list with elements having additional shortcuts pointing towards other elements located further in the list [7]. These shortcuts allow operations to complete in $O(\log n)$ steps in expectation. The drawback of employing shortcuts is however to require additional maintenance each time some data is stored or discarded.

Non-blocking skip lists are increasingly popular alternatives to B-trees in main-memory databases, like *memsql*[1], as they are latch-free and can be traversed in sorted order. By being non-blocking, a skip list ensures that the system as a whole always makes progress. However, in a highly concurrent context the additional maintenance causes contention overheads on existing skip lists [3,4,8] by increasing the probability of multiple *threads* (or processes) interfering on the same shared element. Such contention could translate into performance losses in multicore applications, like in-memory key-value store.

We recently observed a similar issue in concurrent trees that led us to derive a binary search tree algorithm especially suited for transactional memory [2]. Our *contention-friendly* non-blocking skip list demonstrates that these algorithmic concepts can be adapted to improve the performance of a different data structure relying exclusively on compare-and-swap, which makes it inherently non-blocking. In addition, our skip list guarantees the atomicity of insertions, deletions and lookups of key-value pairs as shown in the companion technical report [1]:

Theorem 1. *Each of the contains, insert, delete operations implemented by the contention-friendly non-blocking skip list satisfy linearizability.*

The contention-friendly non-blocking skip list aims at accommodating contention of modern multicore machines. To this end, it exploits a genuine decoupling of each updating access into an eager abstract modification and a lazy and selective structural adaptation.

Eager Abstract Modification. The eager abstract modification consists in modifying the abstraction while minimizing the impact on the skip list itself and returning as soon

[*] A full version is available in [1]. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 238639, ITN project TransForm.

[1] http://developers.memsql.com/docs/1b/indexes.html

as possible for the sake of responsiveness. Existing skip lists typically maintain a precise distribution of nodes per level, hence each time the abstraction is updated, the invariant is checked and the structure is accordingly adapted as part of a single operation. While an update to the abstraction may only need to modify a single location to become visible, its associated structural adaptation is a global modification that could potentially conflict with any concurrent update. In order to avoid these additional conflicts, when a node is inserted in the contention-friendly skip list only the bottom level is modified and the additional structural modification is postponed until later. When an element is removed the operation is separated into a logical deletion marking phase followed by physical removal and garbage collection phases.

Lazy Selective Adaptation. The lazy selective adaptation, which can be deferred until later, aims at adapting the skip list structure to the abstract changes by re-arranging elements or garbage collecting deleted ones. To guarantee the logarithmic complexity of accesses when there is no contention in the system, the structure is adapted by updating the upper levels of the skip list when contention stops.

The structural adaptation is *lazy* because it is decoupled from the abstract modifications and executed by one or multiple independent thread(s). Hence many concurrent abstract modifications may have accessed the skip list while no adaptations have completed yet. We say that the decoupling is *postponed* from the system point of view.

This postponement has several advantages whose prominent one is to enable merging of multiple adaptations in one simplified step: only one traversal is sufficient to adapt the structure after a bursts of abstract modifications. Another interesting aspect is that it gives a chance to insertions to execute faster: if the element to be inserted is marked as logically deleted, then the insertion simply needs to logically insert by unmarking it. This avoids the insertion to allocate a new node and to write its value in memory.

Performance. Our preliminary evaluations on a 24-core machine show that a Java implementation of the contention-friendly non-blocking skip list can improve the performance of one of the mostly used non-blocking skip lists, the JDK adaptation by Lea of the Harris and Michael's lists [5,6], by a multiplying factor of 2.5.

References

1. Crain, T., Gramoli, V., Raynal, M.: A contention-friendly, non-blocking skip list. Technical Report RR-7969, IRISA (May 2012)
2. Crain, T., Gramoli, V., Raynal, M.: A speculation-friendly binary search tree. In: PPoPP (2012)
3. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: PODC (2004)
4. Fraser, K.: Practical lock freedom. PhD thesis. Cambridge University (September 2003)
5. Harris, T.L.: A Pragmatic Implementation of Non-blocking Linked-Lists. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, p. 300. Springer, Heidelberg (2001)
6. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA, pp. 73–82 (2002)
7. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM 33 (June 1990)
8. Sundell, H., Tsigas, P.: Scalable and lock-free concurrent dictionaries. In: SAC (2004)

Brief Announcement: Consensus and Efficient Passive Replication

Flavio Junqueira and Marco Serafini

Yahoo! Research, Barcelona, Spain
{fpj,serafini}@yahoo-inc.com

Passive replication is a popular practical approach to fault tolerance [1]. Using the Paxos consensus protocol [4] to implement it is seeing a growing popularity lately, but requires taking care of peculiar constraints. State updates must be applied using the same sequence of generation: if a primary is in state A and executes an operation making it transition to state B, the resulting state update δ_{AB} must be applied to the state A. Applying it to a different state $C \neq A$ is not safe because it might lead to an incorrect state, which is inconsistent with the history observed by replicas and clients. Paxos does not necessarily preserve the dependency between A and the delivery of δ_{AB}, as observed in [3].

A general approach to implement passive replication on top of consensus is semi-passive replication. With semi-passive replication, replicas execute *sequential*, non-overlapping consensus instances [2]. Our evaluation using Paxos indicates that running pipelined consensus instances, which is the default behavior of Paxos state machine replication, increases saturated system throughput by 60%.

Several practical systems implement passive replication using Paxos together with system-specific serialization techniques that limit concurrency: for example, they may lock resources or abort concurrent and conflicting state updates.

Existing consensus-based implementations of passive replication restrict parallelism. In this work, we consequently seek to answer the question of whether consensus is a fundamental building block for *efficient* passive replication. We answer this question in the positive with three key observations.

First, we observe that using the *primary order atomic broadcast* (*POabcast*) primitive defined by Junqueira *et al.* [3] leads to a straightforward implementation of passive replication that satisfies linearizability. When a process is (unreliably) elected primary by the *POabcast* layer, it starts executing operations tentatively, broadcasting the resulting state updates with *POabcast*. Processes commit state updates when they are delivered by *POabcast*, and reply to clients.

Second, we note that executing sequential consensus instances, as in semi-passive replication, can be seen as a simple way to implement *POabcast*, but this implementation does not admit pipelining.

Third, we derive the first implementation of *POabcast* on top of *pipelined* consensus instances and an Ω leader elector. Our implementation does not need the *explicit* synchronization phase that differentiates Paxos and Zab. In Zab, before becoming a primary and starting to propose new values, a new leader must make sure that a quorum of replicas agree on the sequence of delivered values sent by the previous leaders. Processes participating to this synchronization suspend regular agreement on values. Our implementation achieves synchronization *implicitly* through the use of a sequence of pipelined consensus instances. Consensus instances are never suspended; the same instance can be used by a process to elect itself as a new primary and by another process to agree on a value. This enables using consensus as an underlying building block without modifying the consensus algorithm itself.

We now briefly describe how our algorithm implements the *broadcast* and *deliver* primitives of *POabcast* using the underlying *propose* and *decide* primitives of consensus. When a process p is elected leader by the underlying Ω leader election module, it chooses a unique epoch number e and proposes a (`NEW-EPOCH`, e) value in the smallest consensus instance i where p has not yet reached a decision. If this `NEW-EPOCH` value is decided, we say that e is *established* and p is elected as new primary; in this case, p proposes locally broadcasted values in the next consensus instances, in a pipelined manner, using `WRITE` values. Synchronization upon primary election is implicit: all processes establishing e in consensus instance i have decided and delivered the same sequence of values in the instances preceding i. Processes deliver `WRITE` values of epoch e when they are decided in instances following i, until a different epoch is established.

`WRITE` values of an epoch are not necessarily decided in the same order as they are proposed by the primary. If an epoch e' was established before e, `WRITE` values of e' may be decided in instances following i. This interleaving occurs due to pipelining: the primary of e' may have proposed a value for an instance following i before establishing e in instance i. If a `WRITE` value of e was proposed, but not decided, in an instance $j > i$, it needs to be proposed again by p in a consensus instance $k > j$. Primaries keep track of values that were proposed but not yet decided, and backups keep track of values that have been decided but cannot be delivered due to gaps in the order proposed the primary of e.

Our results show that consensus-based implementations can have the same performance, in stable periods, as direct implementations of *POabcast* that are not based on consensus, like the Zab algorithm [3], at the cost of higher space complexity for keeping out-of-order values.

Acknowledgement. This work has been partially supported by the INNCORPORA - Torres Quevedo Program and the EU SRT-15 project (ICT-257843).

References

1. Budhiraja, N., Marzullo, K., Schneider, F.B., Toueg, S.: The primary- backup approach, pp. 199–216. ACM Press/Addison-Wesley (1993)
2. Défago, X., Schiper, A., Sergent, N.: Semi-passive replication. In: IEEE SRDS, pp. 43–50 (1998)
3. Junqueira, F., Reed, B., Serafini, M.: Zab: High-performance broadcast for primary-backup systems. In: IEEE DSN, pp. 245–256 (2011)
4. Lamport, L.: The part-time parliament. ACM Trans. on Comp. Sys (TOCS) 16(2), 133–169 (1998)

Brief Announcement: Anonymity, Failures, Detectors and Consensus

Zohir Bouzid[1,⋆] and Corentin Travers[2,⋆⋆]

[1] University Pierre et Marie Curie - Paris 6, LIP6, France
`zohir.bouzid@lip6.fr`
[2] University Bordeaux 1, LaBRI, France
`corentin.travers@labri.fr`

Abstract. The paper determines the weakest failure detector for consensus in asynchronous, crash prone and anonymous message passing systems.

Anonymous Systems. A common, often implicit, assumption in distributed computing is that the system is *eponymous*: each process is provided with an unique identifier. On the other hand, in *anonymous* systems, processes have no identity and are programmed identically. When provided with the same input, processes in such systems are indistinguishable. Anonymity adds a new, challenging, difficulty to distributed computing.

Consensus and Failure Detectors. Besides anonymity, a major difficulty is coping with *failures* and *asynchrony*. Many simple problems cannot be solved in asynchronous and failures-prone distributed systems. A prominent example is consensus, which plays a central role in fault-tolerant distributed computing. Informally, n processes, each starting with a private value, are required to agree on one value chosen among their initial values. It is well known that *asynchronous fault tolerant* consensus is impossible even if processes have unique identifiers as soon as at least one process may fail by crashing [7]. Consensus is thus impossible in anonymous, asynchronous and failure-prone message passing system.

A *failure detector* [5] is a distributed device that provides processes with possibly unreliable information about failures. According to the quality of the information, several classes of failure detectors can be defined. Given a distributed problem P, a natural question is to determine the *weakest failure detector* for P, that is a failure detector D which is both *sufficient* to solve the problem – there is an asynchronous, algorithm that uses D to solve P – and *necessary*, in the sense that any failure detector D' that allows solving P can be used to emulate D. For consensus in eponymous system, it has been shown that the combination of failure detectors Ω and Σ is both sufficient and necessary [4,6].

The failure detector based approach has been investigated recently in anonymous systems [1,2]. In particular, [1] presents several anonymous variants of standard failure detectors and study their relative power. Anonymous variants of Ω and Σ, called $A\Omega$ and $A\Sigma$ are defined, and a consensus algorithm based on these two failure detectors is described.

⋆ Supported by DIGITEO project PACTOLE.
⋆⋆ Supported in part by the ANR project DISPLEXITY.

Contributions of the Paper. The paper generalizes the tight bound on failure detection for consensus in eponymous systems of [4,6] to the case of anonymous systems.

It first introduces two new classes of failure detectors, called $A\Omega'$ and $A\Sigma'$, that generalize Ω and Σ and shows that they are both necessary and sufficient to solve consensus in anonymous systems subject to any number of crash failures. $A\Omega'$ eventually distinguishes a *set* L of non-faulty, possibly identical, processes. For processes in L, the output $A\Omega'$ eventually converges to the same value, from which the size of L can be inferred. Each other process is eventually informed that it is not part of this set. The output of $A\Sigma'$ has two components: a label and a set of quorums, each quorum being a multi-set of labels. At each process, both components may never stabilize. Labels may be seen as temporary identifiers assigned to the processes by the failure detector. The label component allows to map each quorums Q to a collection of sets of processes (we call such a set an *instance* of Q). Instances of quorums have similar properties as the set of processes identities output by Σ: every two instances intersect and eventually each quorum has an instance that contains only correct processes. Although similar to the classes $A\Omega$ and $A\Sigma$ introduced in [1], $A\Omega'$ and $A\Sigma'$ are strictly weaker.

An $(A\Sigma' \times A\Omega')$-based consensus algorithm for anonymous and asynchronous system is then presented. The algorithm tolerates an arbitrary number of failures and is "genuinely anonymous" [1], as processes are not required to be aware of the total number n of processes. Finally, the paper shows how to emulate $A\Omega'$ and $A\Sigma'$ from any pair $(\mathcal{A}, D)$, where $\mathcal{A}$ is a consensus algorithm that uses failure detector D. A standard procedure in extracting weakest failure detectors is the construction of a *precedence graph* that describes temporal relationships as well as ownership between failure detector outputs. The main challenge lies in extending the precedence graph construction to the anonymous case. Once this difficulty is resolved, the proof strives to reuse the extraction of Ω and Σ in eponymous systems. See [3] for more details.

References

1. 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)
2. Bonnet, F., Raynal, M.: Consensus in anonymous distributed systems: Is there a weakest failure detector? In: AINA 2010, pp. 206–213. IEEE (2010)
3. Bouzid, Z., Travers, C.: Anonymity, failures, detectors and consensus. Technical report, http://hal.inria.fr/hal-00723309
4. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM 43(4), 685–722 (1996)
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)
6. Delporte-Gallet, C., Fauconnier, H., Guerraouip, R.: Tight failure detection bounds on atomic object implementations. J. ACM 57(4) (2010)
7. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (1985)

Brief Announcement: Do VNet Embeddings Leak Information about ISP Topology?

Yvonne-Anne Pignolet[1], Stefan Schmid[2], and Gilles Tredan[3]

[1] ABB CRC, Switzerland
[2] TU Berlin & T-Labs, Germany
[3] CNRS-LAAS, France

Abstract. This paper initiates the study of adversarial topology inference with virtual network (VNet) embeddings in ISP networks. As an example, we sketch how to infer cactus graphs with VNet request complexity $O(n)$.

Contribution Sketch

An Internet Service Provider's (ISP) network infrastructure properties often constitute a business secret, not only for a competitive advantage, but also because the discovery of, e.g., bottlenecks, may be exploited for attacks or bad publicity. Hence, providers are often reluctant to open the infrastructure to novel technologies and applications that might leak information. We raise the question whether today's trend of *network virtualization* [1], can be exploited to obtain information about the infrastructure. Network virtualization allows customers to request *virtual networks (VNets)* on demand. A VNet defines a set of virtual nodes (e.g., virtual machines) interconnected via virtual links according to the specified VNet topology over a substrate network. In this paper we consider VNet requests which do not impose any location constraints on where the virtual nodes are mapped to. This flexibility in the VNet specification can be used by the operator to optimize the VNet embedding. Thus the VNet can be realized on arbitrary substrate nodes and paths. We assume that as long as a network provider has sufficient resources available to embed a VNet, it will always accept the request. We study how this behavior can be exploited to infer the *full topology* of the substrate.

Model. Our setting comprises two entities: a *customer* (adversary) issuing VNet requests and a *provider* that performs access control and embeddings of VNets (e.g., [2]). We model VNet requests as simple, undirected, weighted graphs $G = (V, E)$ where V denotes the virtual nodes and E denotes the virtual edges connecting nodes in V. Both sets can be weighted to specify requirements, e.g., computation or storage resources at nodes or bandwidth of edges. The infrastructure network (*substrate*) is a weighted undirected graph $H = (V, E)$, with V denoting the substrate nodes, E the substrate links, and the weights describe the capacity of nodes and edges. Without loss of generality, we assume that there are no parallel edges or self-loops either in VNet requests or in the substrate, and that H is connected. In order to focus on topological aspects, we assume the substrate graph elements in H to have a constant capacity of one unit and the requested nodes and links to come with a demand of one unit as well. A virtual link which is mapped to more than one substrate link, i.e., forms a *path*, uses resources of $\epsilon > 0$ at the *relay nodes*, the substrate nodes which do not constitute endpoints of the virtual link and merely serve for forwarding. As a performance measure, we introduce the notion of *request complexity*, i.e., the number of VNet requests which have to be issued until a given network is fully known to the adversary. Thus we study algorithms that "guess" the substrate topology H among the set $\mathcal{H}$ of possible topologies allowed by the model. Given a VNet request G, the provider always responds with an *honest binary reply* informing the customer whether the requested VNet is embeddedable on the substrate. Hence we assume that the provider does not use any means to conceal its network, e.g., by randomizing its binary replies. Based on the reply, the customer may then decide to ask the provider to embed the corresponding VNet G on H, or to continue asking for other VNet embeddings.

Cactus Graph Inference Algorithm. *Cactus graphs* are particularly interesting in the networking context (cf e.g., *Rocketfuel networks*, `www.cs.washington.edu/research/networking/rocketfuel/`). Formally, a cactus is a connected graph in which any two simple cycles have at most one node in common. Or equivalently, every edge in the cactus graph belongs to at most one 2-connected component, i.e., cactus graphs do not contain diamond graph minors. The cactus discovery algorithm CACTUS is based on the idea of incrementally growing the request graph and adding longest sequences of cycles (in our case triangle graphs consisting of three virtual nodes, short: Y) and chains (in our case two connected virtual nodes, short C) recursively. We first try to find the basic "knitting" of the "branches" of the given cactus. Only once such a maximal sequence is found for a branch, the algorithm discovers the detailed structure of the chain/cycle sequence by inserting as many nodes on the chains and cycles as possible. Intuitively, the nodes of a longest sequence of virtual cycles and chains will serve as an "anchor" for extending further branches in future requests: since the sequence is maximal and no more nodes can be embedded, the number of virtual nodes along the sequence must equal the number of substrate nodes on the corresponding substrate path. The endpoints of the sequence thus cannot have any additional neighbors, and we can recursively explore the longest branches of nodes discovered along the sequence.

Theorem 1. CACTUS *discovers any cactus with optimal request complexity* $\Theta(n)$.

Proof Sketch: The correctness of the algorithm follows from the fact that requesting a cyclic "motif" Y saturates a corresponding 2-connected component of the cactus graph, and k consecutive triangles (triangles having one common vertex, Y^k) can only be embedded on ℓ consecutive 2-connected components if $k \leq \ell$ (i.e., they constitute anchors). Once we have identified the maximal k, j such that $(Y^k C)^j$ can be embedded in H, we know that each of these Y^k motifs capture the basic "knitting" of the part of the cactus branch and for each C of this knitting the next requests find the maximal k', j' to replace it by $C(Y^{k'}C)^{j'}$. The time complexity then follows from assigning request costs to the cactus edges. The lower bound and asymptotic optimality is due to the number of possible cactus graphs and the binary reply scheme. □

References

1. Chowdhury, M.K., Boutaba, R.: A survey of network virtualization. Elsevier Computer Networks 54(5) (2010)
2. 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)

Brief Announcement: Efficient Private Distributed Computation on Unbounded Input Streams[⋆]

Shlomi Dolev[1], Juan Garay[2], Niv Gilboa[3], Vladimir Kolesnikov[4], and Yelena Yuditsky[1]

[1] Dept. of Computer Science, Ben-Gurion University of the Negev, Israel
{dolev,yuditsky}@cs.bgu.ac.il
[2] AT&T Labs – Research, Florham Park, NJ
garay@research.att.com
[3] Dept. of Communication Systems Engineering, Ben-Gurion University of the Negev, Israel
niv.gilboa@gmail.com
[4] Bell Laboratories, Murray Hill, NJ
kolesnikov@research.bell-labs.com

We consider a distributed computation setting in which a party, whom we refer to as *the dealer*, has a finite state automaton (FSA) $\mathcal{A}$ with m states, which accepts an (*a priori* unbounded) stream of inputs $x_1, x_2, \ldots$ received from an external source. The dealer delegates the computation to agents $A_1, \ldots, A_n$, by furnishing them with an implementation of $\mathcal{A}$. The input stream $x_1, x_2, \ldots$ is delivered to all agents in a synchronized manner during the online input-processing phase. Finally, given a signal from the dealer, the agents terminate the execution, submit their internal state to the dealer, who computes the state of $\mathcal{A}$ and returns it as output.

We consider an attack model where an entitiy, called the adversary, *Adv*, is able to adaptively "corrupt" agents (i.e., inspect their internal state) during the online execution phase, up to a threshold[1] $t < n$. We do not aim at maintaining the privacy of the automaton $\mathcal{A}$; however, we wish to protect the secrecy of the state of $\mathcal{A}$ and the inputs' history. We note that *Adv* may have external information about the computation, such as partial inputs or length of the input sequence, state information, etc. This auxiliary information, together with the knowledge of $\mathcal{A}$, may exclude the protection of certain configurations, or even fully determine $\mathcal{A}$'s state. We stress that this cannot be avoided in any implementation, and we do not consider this an insecurity. Thus, our goal is to prevent the leakage or derivation by *Adv* of any knowledge from seeing the execution traces which *Adv* did not already possess.

⋆ This research has been supported by the Ministry of Science and Technology, the Institute for Future Defense Technologies Research named for the Medvedi, Shwartzman and Gensler Families, the Israel Internet Association, the Lynne and William Frankel Center for Computer Science at Ben-Gurion University, Rita Altura Trust Chair in Computer Science, *Israel Science Foundation* (grant number 428/11), Cabarnit Cyber Security MAGNET Consortium and MAFAT. Extended version can be found in the Computing Research Repository (CoRR), August 2012.

[1] We note that more general access structures may be naturally employed with our constructions.

Dolev *et al.* [1] were able to provide very strong (unconditional, or information-theoretic) security for computations performed by a finite-state machine (FSA), at the price however of the computation being efficient only for a small set of functions, as in general the complexity of the computation is exponential in the size (number of states) of the FSA computing the function.

In this work, we *minimally*[2] weaken the original model by additionally assuming the existence of one-way functions (and hence consider polynomial-time adversaries—in the security parameter κ), and in return achieve very high efficiency as a function of the size of the FSA. We stress that we still consider computation on *a priori* unbounded number of inputs, moreover, the size of the agents' state is independent of it, and where the online (input-processing) phase incurs *no communication*.

Our work extends the work of [1]. Towards our goal of making never-ending and private distributed computation practical, we introduce an additional (minimal) assumption of existence of one-way functions (and hence pseudo-random number generators [PRGs]), and propose the following constructions:

– A scheme with (n, n) reconstruction (where all n agents participate in reconstruction), where the storage and processing time per input symbol is $O(mn)$ for each agent. The reconstruction complexity is $O(mn)$.
– A scheme with $(t + 1, n)$ reconstruction (where t corrupted agents do not take part in the reconstruction), where the above costs are $O(m\binom{n-1}{t-1})$. [3] The reconstruction complexity is $O(m(t + 1))$.

Regarding tools and techniques, the carefully orchestrated use of PRGs and secret-sharing techniques [2] allows our protocols to hide the state of the computation against an adaptive adversary by using share re-randomization. Typically, in the context of secret sharing, this is simply done by the addition of a suitable (i.e., passing through the origin) random polynomial. However, due to the no-communication requirement, share re-randomization is a lot more challenging in our setting. This is particularly so in the more general case of the $(t + 1, n)$-reconstruction protocol. We achieve share re-randomization by sharing PRG seeds among the players in a manner which allows players to achieve sufficient synchronization of their randomness, which is resilient to t corruptions.

References

1. Dolev, S., Garay, J., Gilboa, N., Kolesnikov, V.: Secret Sharing Krohn-Rhodes: Private and Perennial Distributed Computation. In: ICS (2011)
2. Shamir, A.: How to Share a Secret. CACM 22(11), 612–613 (1979)

[2] Indeed, the existence of one-way functions is considered a minimal assumption in contemporary cryptography. In particular, we do not allow the use of public-key cryptography.

[3] For some values of t, e.g. $t = \frac{n}{2}$, this quantity would be exponential in n. This does not contradict our assumption on the computational power of the participants; it simply means that, given κ, for some values of n and t this protocol cannot be executed in the allowed time.

Brief Announcement: Fast Travellers: Infrastructure-Independent Deadlock Resolution in Resource-restricted Distributed Systems

Sebastian Ertel[1], Christof Fetzer[1], and Michael J. Beckerle[2]

[1] Technische Universität Dresden
Dresden, Germany
firstname.lastname@tu-dresden.de
[2] Waltham, MA, USA
michael.beckerle@alum.mit.edu

Introduction. In the area of data integration and middleware, distributed data processing systems create directed workflows to perform data cleansing, consolidation and calculations before emitting results to targets such as data warehouses. To provide fault tolerance, expensive system-wide checkpoints of distributed workflows want to be performed on the level of seconds while commits to transactional target resources must happen much more frequently to satisfy near real-time result latency [1] and small transaction size requirements. When there exists non-determinism in the workflow, the commit against a transactional target is allowed to be issued only when the determinants were saved to stable storage and deterministic replay can assure exactly-once result delivery. That is, there exists a *dependency*: the process q (a.k.a. operator or component in the context of data integration) executing the transaction is not allowed to make forward progress unless it has received the notification of the non-deterministic process p stating that the results to be committed can be replayed deterministically in the event of a crash.

The Deadlock Problem. Two challenges exist: 1) the limited system view of the processes and 2) their resource limitations. The first challenge requires a process to have no knowledge about the workflow it is contained in; a common distributed system model aspect [2]. Therewith, creating new connections especially for the above dependency is neither favourable nor possible. A solution based on already existing FIFO channels defined and maintained by the system is desirable. Respectively, the distributed algorithm to coordinate the commits, sends the notification, a *marker* m, in-order with the data. But in between p and q, the data stream can be enriched with a theoretically unbounded number of new messages. In contrast to that, the second challenge refers to the fact that a transaction at q is restricted to a maximum size, modelled by input buffer I_q, while process p only has a limited output buffer Q_p to fulfil latency requirements. Hence, it can not be assured that m arrives at q in the interval $|I_q|$.

Fast Travellers. We solve the above Deadlock Problem by extending our system model such that a channel supports out-of-band message transmission, as known from TCP out-of-band. Respectively, we classify markers with respect to their channel transmission characteristics.

– *Slow Travellers (ST)* are markers in the classical sense that travel through the channels in-order with all other messages (as described in the distributed snapshot algorithm [3]).
– *Fast Travellers (FT)* are markers that are transmitted out-of-band with respect to all messages among a channel.

To always enable the receipt of a Fast Traveller, we state that every process leaves one spot available in its input buffer at any time. The solution for our deadlock problem obviously suggests that the marker m has to be a Fast Traveller.

Assuring Correctness. But as a matter of fact, it is essential to the correctness of most marker-based algorithms that the marker actually travels in-order with the data/messages. For example, the deterministic replay algorithm requires that no messages that can not be replayed deterministically are committed to the transactional resource. This reasoning is based on the "happened before" relationship of message arrivals in the distributed snapshot algorithm [3]. There, the receipt of a Slow Traveller at any two processes p and q with state s_p and s_q marks these states as *computationally equivalent*; $s_p \equiv s_q$. We also define the state s_q of process q when the marker was not received yet as *computationally before ($s_q < s_p$)* A distributed algorithm is correct iff the delivery of a message, sent by p after the m was sent in state s_p, is disabled at state s_q, where $s_q \leqq s_p$.

Marker Pairs. Therefore, we combine the two traveller types such that the creation of a marker at process p produces two messages: 1) a Fast Traveller ft to resolve deadlocks and optimize the resource usage among a target process q and 2) a Slow Traveller st to preserve the correctness of the algorithms. Whenever process q receives ft and adds it to I_q, it holds that q's current state $s_q < s_p$ due to transmission of the messages among channel c. Furthermore, it holds that q's subsequent states up until the arrival of st are computationally in the past of s_p and therewith actions in q depending on ft are enabled. The receipt of st, where $s_p \equiv s_q$, only evicts ft from I_q in order to disable actions depending on ft again and assure correctness of the marker algorithm.

We used our Marker Pair Algorithm to efficiently solve the above deadlock problem in our data integration system[1] and are convinced that there exist many more use cases for Fast Travellers in a variety of different distributed algorithms.

References

1. Polyzotis, N., Skiadopoulos, S., Vassiliadis, P., Simitsis, A., Frantzell, N.: Supporting streaming updates in an active data warehouse. In: ICDE (2007)
2. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco (1996)
3. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. 3, 63–75 (1985)

[1] `http://ohua.sourceforge.net`

Brief Announcement: Hashed Predecessor Patricia Trie - A Data Structure for Efficient Predecessor Queries in Peer-to-Peer Systems

Sebastian Kniesburges and Christian Scheideler

Department of Computer Science
University of Paderborn
D-33102 Paderborn
Germany
{seppel,scheideler}@upb.de

Abstract. The design of efficient search structures for peer-to-peer systems has attracted a lot of attention in recent years. In this announcement we address the problem of finding the predecessor in a key set and present an efficient data structure called hashed Predecessor Patricia trie. Our hashed Predecessor Patricia trie supports *PredecessorSearch(x)* and *Insert(x)* in $\mathcal{O}(\log \log u)$ and *Delete(x)* in $\mathcal{O}(1)$ hash table accesses when u is the size of the universe of the keys. That is the costs only depend on u and not the size of the data structure. One feature of our approach is that it only uses the lookup interface of the hash table and therefore hash table accesses may be realized by any distributed hash table (DHT).

1 Introduction

In this brief announcement we consider the predecessor problem in peer-to-peer systems. We present a data structure that efficiently supports the predecessor problem with the help of any common DHT, e.g. Chord or Pastry. We define the predecessor in the following way: Given a key set S with a total order and a search key x, find $\max\{y \in S | y \leq x\}$. We interpret $z \leq x$ as z is lexicographically smaller than x. In the following we only consider binary strings. The predecessor problem has many applications ranging from string matching problems, IP lookup in Internet routers and computational geometry to range queries in distributed file-sharing applications. Our data structure supports the following operations: *Insert(x)*: this adds the key x to the set S. If x already exists in S, it will not be inserted a second time. *Delete(x)*: this removes the key from the set S, i.e., $S := S - \{x\}$. *PredecessorSearch(x)*: this returns a key $y \in S$ that is the predecessor of x. Related data structures include trie hashing [2] and the popular x- and y-fast tries [3] other related work is mentioned in our previous paper [1]. We think that our solution using only a single Patricia trie is intuitive and simple to understand. Furthermore it is applicable to any hash table and thus also DHTs.

2 Our Results

The hashed Predecessor Patricia trie is based on the hashed Patricia we introduced in [1]. This is constructed by adding some additional nodes to the Patricia trie to allow a binary search on the prefix lengths. For details of this construction see [1]. In our extended approach, the hashed Predecessor Patricia trie, we assume that all inserted keys have the same length $\log u$ and modify the hashed Patricia trie by adding some further pointers to enable an efficient predecessor search. All leaves form a sorted doubly-linked list. Differing from the hashed Patricia trie we store for each node v a pointer to the largest key $l_{max}(v)$ in its left subtrie instead of an arbitrary key in its subtries. To ensure efficient updates all the pointers are undirected, i.e. each leaf stores the start nodes of the pointers pointing to it. The basic idea to find the predecessor for a search key x is to use two consecutive binary searches according to [1]. The first binary search finds the node u such that u's identifier is the largest prefix of x among all node identifiers. The second binary search then looks for the ancestor w of u such that $l_{max}(w)$ is the predecessor of x. Each binary search needs $\mathcal{O}(\log\log u)$ *hashtable lookup* (HT-Lookup) operations. By this construction it follows that each inner node stores at most $\mathcal{O}(1)$ pointers, and at most $\mathcal{O}(1)$ pointers point to the same leaf. Then the following theorem holds.

Theorem 1. *An execution of PredecessorSearch(x) needs $\mathcal{O}(\log\log u)$ hashtable lookup (HT-Lookup) operations, an execution of Insert(x) needs $\mathcal{O}(\log\log u)$ HT-Lookup and $\mathcal{O}(1)$ HT-Write operations and an execution of and Delete(x) needs $\mathcal{O}(1)$ HT-Lookup and $\mathcal{O}(1)$ HT-Write operations. The hashed Predecessor Patricia trie needs $\Theta(\sum_{k\in S}\log u)$ memory space, where $\sum_{k\in S}\log u$ is the sum of the bit lengths of the stored keys.*

References

1. Kniesburges, S., Scheideler, C.: Hashed Patricia Trie: Efficient Longest Prefix Matching in Peer-to-Peer Systems. In: Katoh, N., Kumar, A. (eds.) WALCOM 2011. LNCS, vol. 6552, pp. 170–181. Springer, Heidelberg (2011)
2. Ramabhadran, S., Ratnasamy, S., Hellerstein, J.M., Shenker, S.: Brief announcement: prefix hash tree. In: PODC, p. 368 (2004)
3. Willard, D.E.: Log-logarithmic worst-case range queries are possible in space theta(n). Inf. Process. Lett. 17(2), 81–84 (1983)

Brief Announcement: Naming and Counting in Anonymous Unknown Dynamic Networks[⋆]

Othon Michail, Ioannis Chatzigiannakis, and Paul G. Spirakis

Computer Technology Institute & Press "Diophantus" (CTI), Patras, Greece
{michailo,ichatz,spirakis}@cti.gr

Contribution. We study the fundamental naming and counting problems in networks that are anonymous, unknown, and possibly dynamic. Network dynamicity is modeled by the 1-interval connectivity model [KLO10]. We first prove that on static networks with broadcast counting is impossible to solve without a leader and that naming is impossible to solve even with a leader and even if nodes know n. These impossibilities carry over to dynamic networks as well. With a leader we solve counting in linear time. Then we focus on dynamic networks with broadcast. We show that if nodes know an upper bound on the maximum degree that will ever appear then they can obtain an upper bound on n. Finally, we replace broadcast with *one-to-each*, in which a node may send a different message to each of its neighbors. This variation is then proved to be computationally equivalent to a full-knowledge model with unique names.

The Model. A *dynamic network* is modeled by a *dynamic graph* $G = (V, E)$, where V is a static set of n nodes and $E : \mathbb{N}_{\geq 1} \to \mathcal{P}(\{\{u, v\} : u, v \in V\})$ is a function mapping a round number $r \in \mathbb{N}_{\geq 1}$ to a set $E(r)$ of bidirectional links. A dynamic graph/network $G = (V, E)$ is said to be 1-*interval connected*, if, for all rounds $r \in \mathbb{N}_{\geq 1}$, the static graph $G(r)$ is connected [KLO10]. Note that this allows the connected network to change arbitrarily from round to round.

Nodes are *anonymous*, that is they do not initially have any ids, and they do not know the topology or the size of the network, apart from some minimal knowledge when necessary. However, nodes have unlimited local storage. Communication is *synchronous message passing*. We focus on the *one-to-each* message transmission model in which, in every round r, each node u generates a different message $m_{u,v}(r)$ to be delivered to each current neighbor v.

Naming Protocols. We first present a terminating protocol that assigns unique (consecutive if needed) ids to the nodes and informs them of n in $O(n)$-time and then refine the size of its messages. We assume that there is a unique leader l with id 0 (as without it naming is impossible) while all other nodes have id $\perp$.

Main Idea: All already named nodes assign unique ids and acknowledge their id to the leader. All nodes constantly forward all received ids so that they eventually reach the leader. So, at some round r, the leader knows a set of assigned ids $K(r)$. We describe now the termination criterion. If $|K(r)| \neq |V|$ then in at most $|K(r)|$ additional rounds the leader must hear from a node outside $K(r)$. On the other hand, if $|K(r)| = |V|$ no new info will reach the leader in the future and the leader may terminate after the $|K(r)|$-round waiting period ellapses.

⋆ Supported in part by the EU (ESF) and Greek national funds, project FOCUS.

Protocol *Dynamic_Naming.* Initially, every node has three variables $count \leftarrow 0$, $acks \leftarrow \emptyset$, and $latest\_unassigned \leftarrow 0$ and the leader additionally has $latest\_new \leftarrow 0$, $time\_bound \leftarrow 1$, and $known\_ids \leftarrow \{0\}$. A node with $id \neq \perp$ for $1 \leq i \leq k$ sends *assign* $(id, count + i)$ message to its ith neighbor and sets $count \leftarrow count + k$. In the first round, the leader additionally sets $known\_ids \leftarrow \{0, (0,1), (0,2), \ldots, (0,k)\}$, $latest\_new \leftarrow 1$, and $time\_bound \leftarrow 1 + |known\_ids|$. Upon receipt of l *assign* messages (rid_j), a node with $id = \perp$ sets $id \leftarrow \min_j\{rid_j\}$ (in number of bits), $acks \leftarrow acks \cup id$, sends an *ack* $(acks)$ message to all its k current neighbors, for $1 \leq i \leq k$ sends *assign* $(id, count + i)$ message to its ith neighbor, and sets $count \leftarrow count + k$. Upon receipt of l *ack* messages $(acks_j)$,a nonleader sets $acks \leftarrow acks \cup (\bigcup_j acks_j)$ and sends *ack* $(acks)$. A node with $id = \perp$ sends *unassigned* $(current\_round)$. Upon receipt of $l \geq 0$ *unassigned* messages (val_j), a node with $id \notin \{0, \perp\}$ sets $latest\_unassigned \leftarrow \max\{latest\_unassigned, \max_j\{val_j\}\}$ and sends *unassigned* $(latest\_unassigned)$. Upon receipt of l *ack* messages $(acks_j)$, the leader if $(\bigcup_j acks_j) \setminus known\_ids \neq \emptyset$ sets $known\_ids \leftarrow known\_ids \cup (\bigcup_j acks_j)$, $latest\_new \leftarrow current\_round$ and $time\_bound \leftarrow current\_round + |known\_ids|$ and upon receipt of l *unassigned* messages (val_j), it sets $latest\_unassigned \leftarrow \max\{latest\_unassigned, \max_j\{val_j\}\}$. If, at some round r, it holds at the leader that $r > time\_bound$ and $latest\_unassigned < latest\_new$, the leader sends a *halt* $(|known\_ids|)$ message for $|known\_ids| - 1$ rounds and then outputs id and halts. Any node that receives a *halt* (n) message, sends *halt* (n) for $n - 2$ rounds and then outputs id and halts.

A drawback of *Dynamic_Naming* is its $\Theta(n^2)$ bits/message. We now refine it to reduce the message size to $\Theta(\log n)$ paying in $O(n^3)$ termination-time.

Protocol *Individual_Conversations* [Main Idea]. To reduce the size of the messages (i) the assigned names are now of the form $k \cdot d + id$, where id is the id of the node, d is the number of *unique consecutive* ids that the leader knows so far, and $k \geq 1$ is a name counter (ii) Any time that the leader wants to communicate to a remote node that has obtained a unique id it sends a message with the id of that node and a timestamp equal to the current round. The timestamp allows all nodes to prefer this message from previous ones so that the gain is twofold: the message is delivered and no node ever issues a message containing more than one id. The remote node then can reply in the same way. For the assignment formula to work, nodes that obtain ids are not allowed to further assign ids until the leader freezes all named nodes and reassigns to them unique consecutive ids. During freezing, the leader is informed of any new assignments by the named nodes and terminates if all report that no further assignments were performed. A full version of this paper is available at http://arxiv.org/abs/1208.0180 [MCS12].

References

[KLO10] Kuhn, F., Lynch, N., Oshman, R.: Distributed computation in dynamic networks. In: Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, pp. 513–522. ACM (2010)

[MCS12] Michail, O., Chatzigiannakis, I., Spirakis, P.G.: Naming and counting in anonymous unknown dynamic networks. CoRR, abs/1208.0180 (2012)

Brief Announcement: SplayNets
Towards Self-Adjusting Distributed Data Structures

Stefan Schmid[1], Chen Avin[2], Christian Scheideler[3],
Bernhard Haeupler[4], and Zvi Lotker[2]

[1] TU Berlin & T-Labs, Germany; [2] BGU, Israel; [3] U. Paderborn, Germany; [4] MIT, USA

Abstract. This paper initiates the study of self-adjusting distributed data structures or networks. In particular, we present *SplayNets*: a binary search tree based network that is self-adjusting to the routing requests. We derive entropy bounds on the amortized routing cost and show that our splaying algorithm has some interesting properties.

1. Distributed Splay Trees. In the mid 80s, Sleator and Tarjan [1] introduced an appealing new paradigm to design efficient data structures: rather than optimizing traditional metrics such as the search tree depth in the *worst-case*, the authors proposed to make data structures *self-adjusting* and considered the *amortized cost* as the performance metric—the "average cost" per operation for a given sequence s of lookups. The authors described *splay trees*, self-adjusting binary search trees in which frequently accessed elements are moved closer to the root, improving the average access times *weighted by the elements' popularity*. The popularity distribution must not be known in advance and may even change over time. We, in this paper, initiate the study of a *distributed generalization* of splay trees as a network. We consider a distributed data structure, e.g., a structured peer-to-peer (p2p) system or Distributed Hash Table (DHT), where nodes (i.e., "peers") that communicate more frequently should become topologically closer to each other (i.e., reducing the routing distance). This contrasts with most of today's structured peer-to-peer overlays whose topology is often optimized in terms of static global properties only, such as the node degree or the longest routing path.

2. Model and Problem Definition. Given an arbitrary and unknown pattern of communication (or routing) requests σ between a set of nodes $V = \{1, \ldots, n\}$, we attend to the problem of finding good *communication networks* G out of a family of *allowed networks* $\mathcal{G}$. Each topology $G \in \mathcal{G}$ is a graph $G = (V, E)$, and we define a set of *local transformations* on graphs in $\mathcal{G}$ to transform one member $G' \in \mathcal{G}$ to another member $G'' \in \mathcal{G}$. We seek to adapt our topologies smoothly over time, i.e., a changing communication pattern leads to "local" changes of the communication graph over time. We focus on the special case where $\mathcal{G}$ is the set of *binary search trees* (BST), henceforth simply called *BST networks*. Besides their simplicity, such networks are attractive for their low node degree and the possibility to route locally: given an destination identifier (or address), each node can decide locally whether to forward the packet to its left child, its right child, or its parent. The local transformations of BST networks are called *rotations*. Let $\sigma = (\sigma_0, \sigma_1 \ldots \sigma_{m-1})$ be a sequence of m *communication requests* where $\sigma_t = (u, v) \in V \times V$ denotes that a packet needs to be sent from a *source* u to a *destination* v. The cost of the network transformations at time t is denoted by $\rho(\mathcal{A}, G_t, \sigma_t)$ (or simply ρ_t) and captures the number of rotations performed to change G_t to G_{t+1}. We denote with $\mathrm{d}_G(\cdot)$ the distance function between nodes in G, i.e., for two nodes $v, u \in V$ we define $\mathrm{d}_G(u, v)$ to be the number of edges of a *shortest* path between u and v in G. For a given sequence of communication requests, the cost for an algorithm is given by the number of transformations and the distance of the communication requests plus one. For an algorithm $\mathcal{A}$ and given an initial network G_0 with node distance function $\mathrm{d}(\cdot)$ and a sequence $\sigma = (\sigma_0, \sigma_1 \ldots \sigma_{m-1})$ of communication requests over time, we define the *(average) cost* of $\mathcal{A}$ as: *Cost*$(\mathcal{A}, G_0, \sigma) = \frac{1}{m}\sum_{t=0}^{m-1} (\mathrm{d}_{G_t}(\sigma_t) + 1 + \rho_t)$. The *amortized cost* of $\mathcal{A}$ is defined as the worst possible cost of $\mathcal{A}$, i.e., $\max_{G_0,\sigma}$ *Cost*$(\mathcal{A}, G_0, \sigma)$.

3. SplayNets: Algorithm and Analysis. The main idea of our *double splay* algorithm DS is to perform splay tree operations in *subtrees* covering the different communication partners. Concretely, consider a communication request (u, v) from node u to node v, and let $\alpha_T(u, v)$ denote the least common ancestor of u and v in the current BST network T. Furthermore, for an arbitrary node x, let $T(x)$ denote the subtree rooted at x. The formal listing of DS is shown in Algorithm 1: When a request (u, v) occurs, DS first simply splays u to the least common ancestor $\alpha_T(u, v)$ of u and v, using the classic splay operations Zig, ZigZig, ZigZag from [1]. Subsequently, the idea is to splay the destination node v to the child of the least common ancestor $\alpha_{T'}(u, v)$ of u and v in the resulting tree T'. The communication cost of DS can be upper bounded by the empirical entropy of the sources and destinations of the requests. We can also provide a lower bound for any BST network based on the conditional empirical entropy.

Algorithm 1 Double Splay Algorithm DS

```
1: (* upon request $(u, v)$ in $T$ *)
2: $w := \alpha_T(u, v)$
3: $T' :=$ splay $u$ to root of $T(w)$
4: splay $v$ to the child of $T'(u)$
```

Fig. 1. Double Splay Algorithm DS

Theorem 1. *Let σ be an arbitrary sequence of communication requests, then for any initial BST T_0,* Cost$(DS, T_0, \sigma) \in O(H(\hat{X}) + H(\hat{Y}))$ *where $H(\hat{X})$ and $H(\hat{Y})$ are the empirical entropies of the sources and the destinations in σ, respectively.*

Theorem 2. *Given a request sequence σ, for any optimal BST network T:* Cost$(\perp, T, \sigma) \in \Omega(H(\hat{Y}|\hat{X}) + H(\hat{X}|\hat{Y}))$.

A simple corollary of the above results can be obtained when σ follows a *product distribution* (i.e., $H(\hat{X}|\hat{Y}) = H(\hat{X})$): DS is asymptotically optimal if σ follows a product distribution. In our full paper, we will show that DS features several other desirable properties and that it is optimal in some special cases like when σ forms a *laminated set* or a *BST*. In addition we extend Theorem 2 to give more sophisticated lower bounds.

4. Discussion. We regard our work as a first step towards the design of novel distributed data structures and networks which adapt dynamically to the demand.

Reference

1. Sleator, D., Tarjan, R.: Self-adjusting binary search trees. JACM 32(3), 652–686 (1985)

Brief Announcement: Semantics of Eventually Consistent Replicated Sets [⋆]

Annette Bieniusa[1], Marek Zawirski[1,2], Nuno Preguiça[3,1], Marc Shapiro[1], Carlos Baquero[4], Valter Balegas[3], and Sérgio Duarte[3]

[1] INRIA/LIP6, Paris, France
[2] UPMC, Paris, France
[3] CITI, Universidade Nova de Lisboa, Portugal
[4] HASLab, INESC Tec and Universidade do Minho, Portugal

This paper studies the semantics of sets under eventual consistency. The set is a pervasive data type, used either directly or as a component of more complex data types, such as maps or graphs. Eventual consistency of replicated data supports concurrent updates, reduces latency and improves fault tolerance, but forgoes strong consistency (e.g., linearisability). Accordingly, several cloud computing platforms implement eventually-consistent replicated sets [2,4].

The sequential semantics of a set are well known, and are defined by individual updates, e.g., $\{\mathsf{true}\}add(e)\{e \in S\}$ (in "{pre-condition} computation {post-condition}" notation), where S denotes its abstract state. However, the semantics of concurrent modifications is left underspecified or implementation-driven.

We propose the following *Principle of Permutation Equivalence* to express that concurrent behaviour conforms to the sequential specification: *"If all sequential permutations of updates lead to equivalent states, then it should also hold that concurrent executions of the updates lead to equivalent states."* It implies the following behavior, for some updates u and u':

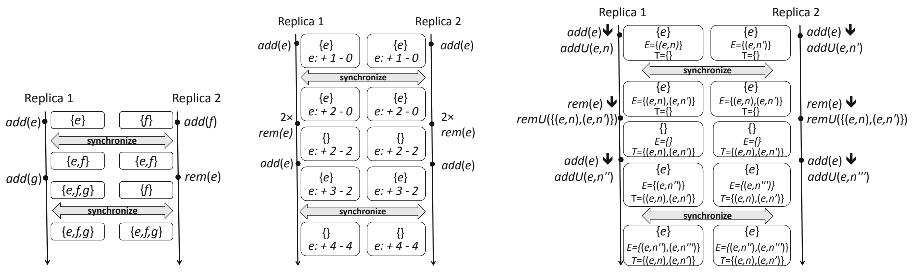
$$\{P\}u; u'\{Q\} \wedge \{P\}u'; u\{Q'\} \wedge Q \Leftrightarrow Q' \quad \Rightarrow \quad \{P\}u \parallel u'\{Q\}$$

Specifically for replicated sets, the Principle of Permutation Equivalence requires that $\{e \neq f\}add(e) \parallel remove(f)\{e \in S \wedge f \notin S\}$, and similarly for operations on different elements or idempotent operations. Only the pair $add(e) \parallel remove(e)$ is unspecified by the principle, since $add(e); remove(e)$ differs from $remove(e); add(e)$. Any of the following post-conditions ensures a deterministic result:

$\{\bot_e \in S\}$	– Error mark
$\{e \in S\}$	– *add* wins
$\{e \notin S\}$	– *remove* wins
$\{add(e) >_{\mathsf{CLK}} remove(e) \Leftrightarrow e \in S\}$	– Last Writer Wins (LWW)

where $<_{\mathsf{CLK}}$ compares unique clocks associated with the operations. Note that

[⋆] This research is supported in part by ANR project ConcoRDanT (ANR-10-BLAN 0208), by ERDF, COMPETE Programme, by Google European Doctoral Fellowship in Distributed Computing received by Marek Zawirski, and FCT projects #PTDC/EIA-EIA/104022/2008 and #PTDC/EIA-EIA/108963/2008.



(a) Dynamo shopping cart (b) C-Set (c) OR-Set

Fig. 1. Examples of anomalies and a correct design

not all concurrency semantics can be explained as a sequential permutation; for instance no sequential execution ever results in an error mark.

A Study of Existing Replicated Set Designs. In the past, several designs have been proposed for maintaining a replicated set. Most of them violate the Principle of Permutation Equivalence (Fig. 1). For instance, the Amazon Dynamo shopping cart [2] is implemented using a register supporting *read* and *write* (assignment) operations, offering the standard sequential semantics. When two *write*s occur concurrently, the next *read* returns their union. As noted by the authors themselves, in case of concurrent updates even on unrelated elements, a *remove* may be undone (Fig. 1(a)).

Sovran et al. and Asian et al. [4,1] propose a set variant, C-Set, where for each element the associated *add* and *remove* updates are counted. The element is in the abstraction if their difference is positive. C-Set violates the Principle of Permutation Equivalence (Fig. 1(b)). When delivering the updates to both replicas as sketched, the add and remove counts are equal, i.e., e is not in the abstraction, even though the last update at each replica is $add(e)$.

Shapiro et al. propose a replicated set design, called OR-Set, [3] that ensures that concurrent *add*/*remove* operations commute. Unlike the others, it satisfies the Principle of Permutation Equivalence, as illustrated in Figure 1(c). Hidden unique tokens distinguish between different invocations of *add*, making it possible to to precisely track which *add* operations are affected by a *remove*.

References

1. Aslan, K., Molli, P., Skaf-Molli, H., Weiss, S.: C-Set: a commutative replicated data type for semantic stores. In: Int. W. on REsource Discovery, RED (2011)
2. DeCandia, G., Hastorun, D., et al.: Dynamo: Amazon's highly available key-value store. In: Symp. on Op. Sys. Principles, SOSP (2007)
3. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-Free Replicated Data Types. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 386–400. Springer, Heidelberg (2011)
4. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: Symp. on Op. Sys. Principles, SOSP (2011)

Brief Announcement: Decoupled and Consensus-Free Reconfiguration for Fault-Tolerant Storage

Eduardo Alchieri[1], Alysson Bessani[2], Fabíola Greve[3], and Joni Fraga[4]

[1] University of Brasília - Brazil
[2] University of Lisbon - Portugal
[3] Federal University of Bahia - Brazil
[4] Federal University of Santa Catarina - Brazil

1 Motivation and Prior Work

Quorum systems are constructions used to ensure consistency and availability of data stored in replicated servers. These systems usually comprise a static set of servers that provide a fault-tolerant read/write (r/w) register accessed by a set of clients. This approach is not adequate for long lived systems since, given a sufficient amount of time, there might be more faulty servers than the threshold tolerated, affecting the system correctness. Moreover, this approach does not allow a system administrator to deploy new machines or replace old ones at runtime and cannot be applied in many systems where, by their very nature, the set of processes that compose the system changes during its execution.

Reconfiguration is the process of changing the set of nodes that comprise the system. Previous works proposed solutions for reconfigurable storage by implementing dynamic quorum systems [2, 3], which rely on consensus for reconfigurations in a way that processes agree on the set of servers to represent the system. However, consensus is not solvable in asynchronous environments and atomic shared memory emulation can be implemented without agreement in static asynchronous systems. Recently, Aguilera et al [1] showed that it is possible to implement reconfigurations without agreement presenting *DynaStore*, a set of algorithms that implements dynamic storage and does not rely on consensus for reconfigurations, that are strongly tied with the r/w protocols. It means that DynaStore r/w protocols explicitly deal with reconfigurations.

2 Contributions

We propose FreeStore, a consensus-free system that implements a fault-tolerant atomic and wait-free register, allowing the servers set (system view) to change at runtime. When compared with DynaStore [1], FreeStore implements a rather different reconfiguration protocol more simple, modular and completely decoupled from r/w protocols.

View generator. FreeStore makes use of a new abstraction called *view generator*. This abstraction aims to capture the agreement requirements of the reconfiguration protocol, being distributed oracles used by servers to generate new system views. Each view installed in the system has an associated view generator to generate a succeeding view.

Definition 1 (View Generator). *A view generator is defined by the following properties:*

– ***Accuracy:*** *a view generator associated with a view v produces the same view w in all correct processes.*
– ***Termination:*** *a started view generator eventually generates a new view.*
– ***Non-triviality:*** *for each view w generated by the view generator associated with a view v, w is more updated than v.*

Non-triviality must be satisfied by any view generator to ensure that updated views are always generated. The satisfaction of Accuracy (A) and Termination (T) properties lead to the definition of four classes of view generators: Perfect ($\mathcal{P}$) – satisfies A and T; Live ($\mathcal{L}$) – satisfies T; Strong ($\mathcal{S}$) – satisfies A; and Weak ($\mathcal{W}$) – does not satisfy neither A or T. FreeStore reconfiguration protocol implements safe and live reconfigurations using a generator as weak as $\mathcal{L}$, a consensus-free generator that does not satisfy Accuracy, i.e., it can generate different views at different processes.

FreeStore *reconfiguration.* FreeStore is composed by two *decoupled* protocols. The reconfiguration protocol is *modular* and implemented through the periodically (assuming that there are reconfiguration requests to be processed) starting the view generator associated with the system current view.

The protocol ensures the system convergence to a single sequence of installed views, even using $\mathcal{L}$. The key idea of the protocol is to choose only some of the generated views produced by $\mathcal{L}$ and define a *sequence of (consistent) views* to be installed until a single final view containing all reconfiguration requests is activated. Since this protocol does not rely on consensus for agreement on a single sequence of views, two or more sequences may be generated in different servers. However, the quorum intersection property ensures that any generated sequence will be a subsequence of any other posterior sequence, what is enough to preserve the reconfiguration safety properties. It is important to mention that FreeStore reconfiguration protocol works with any view generator class.

R/w protocol. The reconfiguration protocol is completely independent from the r/w protocol employed for accessing the register. In consequence, any classical static r/w protocol can be adapted to be used with the system.

References

[1] Aguilera, M., Keidar, I., Malkhi, D., Shraer, A.: Dynamic Atomic Storage Without Consensus. Journal of the ACM 58(2) (2011)
[2] Gilbert, S., Lynch, N., Shvartsman, A.: RAMBO: A Robust, Reconfigurable Atomic Memory Service for Dynamic Networks. Distributed Computing 23(4) (2010)
[3] Martin, J., Alvisi, L.: A Framework for Dynamic Byzantine Storage. In: Proc. of the 34th International Conference on Dependable Systems and Networks (2004)

Brief Announcement: Atomic Consistency and Partition Tolerance in Scalable Key-Value Stores

Cosmin Arad, Tallat M. Shafaat, and Seif Haridi

KTH Royal Institute of Technology, Sweden

Abstract. We propose *consistent quorums* to achieve linearizability in scalable and self-organizing key-value stores based on consistent hashing.

Key-value stores based on consistent hashing [5] provide scalable and self-organizing storage for modern web applications. Simply applying quorum-based read and write operations [3] within replication groups dictated by consistent hashing, fails to achieve atomic consistency in a partially synchronous system prone to network partitions [2]. Given the advantages of consistent hashing (simplicity, incremental scalability, self-organization) and the realities of data-center environments (partial synchrony, node dynamism, and the possibility of network partitions) we set out to achieve linearizability in a dynamic and scalable key-value storage system governed by consistent hashing. We apply results from reconfigurable atomic storage, within dynamic replication groups where node membership is automatically dictated by the principle of consistent hashing.

A naïve approach to achieving consistency is to use quorum-based *read/write* operations within every replication group. This will not work as false failure suspicions, along with consistent hashing, may lead to non-intersecting quorums [2]. Any quorum-based algorithm, such as ABD [3] and Paxos [6], will suffer from the problem of non-intersecting quorums when used with consistent hashing. We propose *consistent quorums* as a solution. Each node has a *view* $\langle v, i \rangle$, where v is the set of nodes in the replication group and i is the version number of the view. A consistent quorum is a quorum of nodes that are in the same view when the quorum is assembled. When a node replies to a request it stamps its reply with its current view. A quorum-based operation (e.g. ABD, Paxos) will succeed only if it finds a quorum of nodes with the same view, i.e., a consistent quorum.

Changes to replication group $\langle v, i \rangle$ are proposed as a new view $\langle v', i+1 \rangle$ via Paxos augmented with consistent quorums. The decision is *installed* on all nodes in $v \cup v'$. In spite of reconfigurations, it can be shown that any two consistent quorums will always intersect [2]. This implies that ABD writes and reads will satisfy linearizability in the presence of reconfigurations and concurrent reconfiguration operations will be applied in a total order [2].

Paxos and ABD are intrinsically partition-tolerant; since they depend on quorums, operations in any partition that contains a quorum will succeed. To maintain partition tolerance when Paxos and ABD are applied within consistent hashing replication groups, we use consistent quorums. We employ a ring unification algorithm [7] that repairs the ring topology of consistent hashing, hence fixing node responsibilities, after a transient network partition. This enables our overall key-value store to be tolerant to network partitions.

A linearizable *read/write* register in a fully asynchronous system model was implemented by the ABD algorithm [3] which used majority replication within a static set of nodes. Atomic registers were extended to dynamic networks with protocols like RAMBO and RDS [4] which used consensus to coordinate the sequence of system reconfigurations. DynaStore [1] showed that reconfigurable atomic registers can be implemented without consensus in a fully asynchronous system. To enable linearizability at large scale, we turn every consistent hashing replication group into a set of reconfigurable atomic registers, one for each key-value object, maintaining a consistent mapping of objects to replication groups.

Similar to previous work on reconfigurable atomic storage [4,1], our approach decouples reconfiguration from register operations, allowing operations to execute concurrently with reconfigurations. While in RDS and DynaStore all operations are forced to contact quorums in all active configurations, with consistent quorums, operations optimistically contact only a single quorum. Only operations that get different consistent quorums between their read and write phases need to repeat the read phase in the new configuration.

Like in RDS [4], we use consensus only for reconfiguration, but more generally, consistent quorums could be used to transform any static quorum-based protocol to be dynamically reconfigurable, while paying for consensus only on reconfiguration, and otherwise maintaining the original protocol's complexity.

The design, implementation, and evaluation of a scalable key-value store based on consistent hashing and consistent quorums, as well as the algorithms and correctness proofs are available in a separate technical report [2].

References

1. Aguilera, M.K., Keidar, I., Malkhi, D., Shraer, A.: Dynamic atomic storage without consensus. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, PODC 2009, pp. 17–25. ACM, New York (2009)
2. Arad, C., Shafaat, T., Haridi, S.: CATS: Linearizability and partition tolerance in scalable and self-organizing key-value stores. SICS Technical Report T2012:04
3. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM 42, 124–142 (1995)
4. Chockler, G., Gilbert, S., Gramoli, V., Musial, P.M., Shvartsman, A.A.: Reconfigurable distributed storage for dynamic networks. J. Parallel Distrib. Comput. 69, 100–116 (2009)
5. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, STOC 1997, pp. 654–663. ACM, New York (1997)
6. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. 16, 133–169 (1998)
7. Shafaat, T., Ghodsi, A., Haridi, S.: Dealing with bootstrapping, maintenance, and network partitions and mergers in structured overlay networks. In: Sixth IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2012. IEEE Computer Society, Washington, DC (2012)

Brief Announcement: Weighted Partial Message Matching for Implicit Multicast Systems[⋆]

William Culhane[1], K.R. Jayaram[2], and Patrick Eugster[1]

[1] Purdue University
{wculhane,peugster}@purdue.edu
[2] HP Labs
jayaramkr@hp.com

In *implicit multicast*, receiving processes delineate the messages they wish to receive by specifying predicates, also called *filters*, on the message's content. In *weighted partial* message matching, distributed applications using implicit multicast protocols do not require that a message match all the elementary constraints constituting the filter. Typically, receiving processes in such applications associate a non-negative weight to each constraint and require that the *match score*, i.e., sum of the weights of matching constraints, exceeds a threshold value. In this paper, we consider top-k weighted partial matching, where a process is interested in multicasting a message only to $k < n$ other processes corresponding to the top-k match scores. This is a fundamental problem underlying online advertising platforms, mobile social networks, online dating, etc.

A message m is a set of attribute/interval pairs $\{a_1 : [v_1, v_1'], \ldots, a_k : [v_k, v_k']\}$. We consider filters to be *conjunctions* of interval constraints, each of which has an associated weight. A filter with n constraints on n attributes is represented as a predicate $\phi = a_1 \in [v_1, v_1'] : w_1 \wedge \ldots \wedge a_n \in [v_n, v_n'] : w_n$. Interval filters are as expressive as regular filters, e.g., `x>1000` where `x` is an integer attribute can be expressed as `x`$\in$ [`1001`, `MAX_INT`] and equalities like `x=1000` can be modeled as `x`$\in$ [`1000`, `1000`]. We use interval filters because they increase efficiency (as we demonstrate in this paper). Given a message m and a filter $\phi = \bigwedge_{i=1}^{n} \delta_i : w_i$, where $\delta_i = a_i \in [v_i, v_i']$, the match score $score(\phi, m) = \sum_{i=1}^{m} w_i \mid \delta_i(m) = \text{TRUE}$. Given a set of filters Θ, and a message m, the top-k weighted partial matching set $\Phi \subseteq \Theta$ is defined as $\Phi = \{\phi \mid score(\phi, m) > 0 \wedge score(\phi, m) > score(\phi', m) \ \forall\ \phi' \in \Theta \setminus \Phi\}$ and $|\Phi| = k$.

The key strategy adopted by our matching algorithm is to consider filters *per attribute*. A broker in an implicit multicast system receives filters from senders and other brokers connected to it. We assume that an incoming filter from client c_i is uniquely identifiable by an identifier denoted by fid. We consider the cases where weights are (1) specified by receivers and are associated with the elementary constraints of a filter, and (2) specified by the sender and attached to the message overriding any weights associated by receivers with filters (shown in Line 24 of Figure 1).

Our algorithm uses a two-level index for filters. Constraints on attributes are indexed at the attribute-level in interval trees, which are in turn indexed by

⋆ Funded by US NSF grants 0644013 and 0834529, DARPA grant N11AP20014, and by Purdue Research Foundation grant 205434.

attribute name in a "master index" hash map (Line 1). An incoming filter is split by attribute name. The constraint for each attribute is added to its respective interval tree (Line 7). If no interval tree exists for an attribute, a new interval tree with the new constraint is created for that attribute (Line 9) and inserted into the master index (Line 11). Removing filters requires removing constraints from their interval trees and removing empty trees from the master index.

Our algorithm tracks filters' match scores via a hash map called *smap* (Line 13). For matching, it retrieves the identifiers and weights for each attribute's filter matching constraints (Line 18). The filters' aggregate scores are updated in *smap* by adding the weight of the matched constraint (Lines 22 and 24). If senders specify weights on attributes, the weights retrieved from the interval tree are discarded (Line 24). After calculating the scores using all filters on the message, the entries in *smap* are inserted into the tree set *topscores* for pruning the top-k. Entries from *smap* are inserted into *topscores* ordered by their scores, and *topscores* is maintained so that its size never exceeds k (Lines 25–33).

Our companion technical report [1] presents our model, algorithm and related work in more detail. We prove that the time complexity of our algorithm for top-k weighted partial matching based on interval trees is $O(M\,log\,N + S\,log\,k)$ and space complexity is $O(M\;N + k)$ where N is the number of filters, M is the number of attributes in a message and S is the number of matching constraints.

```
 1: $mIndex \leftarrow$ new hashmap
 2: upon RECEIVE($\phi$) do
 3:   $fid \leftarrow$ new filter id
 4:   for $(a_i \in [v_i, v'_i] : w_i) \in \phi$ do
 5:     $tree_i \leftarrow$ HMAP-GET($mIndex, a_i$)
 6:     if $tree_i \neq \perp$ then
 7:       INSERT($tree_i, [v_i, v'_i], w_i, fid$)
 8:     else
 9:       $tree_i \leftarrow$new interval tree
10:       INSERT($tree_i, [v_i, v'_i], w_i, fid$)
11:       HMAP-PUT($mIndex, a_i, tree_i$)

12: upon RECEIVE(MSG) do
13:   $smap \leftarrow$ new hashmap
14:   $topscores \leftarrow$ new treeset
15:   $min \leftarrow 0$
16:   for $a_i : [v_i, v'_i] : w_i \in$ MSG do
17:     $tree_i \leftarrow$ HMAP-GET($master, a_i$)
18:     $matches \leftarrow$ INTERSECT($tree_i, [v_i, v'_i]$)
19:     if $w_j = 0 \;\forall\; a_j \in$ MSG then
20:       for all $\langle fid, [v_r, v'_r], w_r \rangle \in matches$ do
21:         $score \leftarrow$ HMAP-GET($smap, fid$)
22:         HMAP-PUT($smap, fid, score + w_r$)
23:     else
24:       HMAP-PUT($smap, fid, score + w_i$)
25:   for all $fid \in$ GET-ALL-KEYS($smap$) do
26:     if SIZEOF($topscores$) $< k$ then
27:       TREESET-ADD($topscores, fid, w$)
28:       if $w < min \vee min = 0$ then
29:         $min \leftarrow w$
30:     else if $min < w$ then
31:       TREESET-REMOVE-MIN($topscores$)
32:       TREESET-ADD($topscores, fid, w$)
33:       $min \leftarrow$ TREESET-FIND-MIN($topscores$)
34:   for all $fid \in$ GET-KEYS($topscores$) do
35:     SEND(MSG) to RECEIVER-OF($fid$)
```

Fig. 1. Algorithm for weighted partial matching of messages to filters

[1] Purdue Computer Science TR # 12-009. See http://docs.lib.purdue.edu/cstech/

Author Index