12-1-2011

Practical, scalable algorithms for Byzantine agreement

Olumuyiwa Oluwasanmi

Olumuyiwa Oluwasanmi

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Jared Saia, Chairperson

Valerie King

Patrick Bridges

Cristopher Moore

Practical, Scalable Algorithms for Byzantine agreement

by

Olumuyiwa Oluwasanmi

B.Sc., Computer Science and Mathematics O.A.U., Ile-Ife, 2000
M.S., Computer Science, University of New Mexico, 2008
M.S., Mathematics, University of New Mexico, 2009

DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor Of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2011

Dedication

To my parents, Hector and Eunice Oluwasanmi, whose vision and efforts made all this possible. To my older siblings Adesola, Adeoti, Ebunoluwa, Adebolanle, Temitope their respective husbands and children for being such a great encouragement, and to my younger brother Ayoade. It's a great blessing to be from a large family. Oluwa ku ise, Jesu seun!

Acknowledgments

I would like to thank my adviser, Jared Saia, for his support, for being such a fantastic mentor, to whom I am incredibly indebted. For his keen insight into solving problems and having the knack for which problems are important, without whose encouragement and support this dissertation would not have been completed. My gratitude also goes to Patrick Bridges for supporting me in the Fall/Spring of 2008/2009. I would like to thank my wife Candace for enduring my working late practically every night, being gracious by proofreading some of my manuscript, thanks for all the love and support in a very difficult year. To the other members of my committee Patrick Bridges, Cris Moore and Valerie King for helpful comments and suggestions that have improved the overall quality of this dissertation.

To Quintin and Terisita Ndibongo, for the gift of their friendship and connecting me with other international students. To Rev. Don Wilson and his wife for hosting me when I first arrived in New Mexico. To Don and Suzanne Templeton and for being such a great help. To Dan and Trudy McGregor for supporting Candace and I before we were married along with Dave Bruskas and his family as well you are all a great encouragement. To Drs Imoh and Gloria Obioh, Dr A.P. Akinola, Dr A. Oluwaranti,Dr Timothy Adagunodo and the late Femi Ninan for encouraging a budding scientist. Steven Onherime for giving me the opportunities at Future Shock that played no small part in my starting the PhD program at New Mexico.

To Navin for many interesting rides and discussions about what is important in Computer Science. To Amitabh Treehan whose enthusiasm is infectious. To Jeff Knockel for very interesting conversations about how awesome C++ is. To Madalitso Muthiya, Patrick Mapalo, Cia Hell, Jacob Agola, George Kyei, Abdissa Zerai, Antoinette Tiopi, Cleophas and Lorenzo Ireland for being friends when I needed it the most.

Prov 18:24 “A man that has friends must show himself friendly: and there is a friend that sticks closer than a brother”. I will be remiss without mentioning you guys for the past 17 years and more who have had my back even though they are thousands of miles away right now. In no paticular order; Bolaji Adeola, Rufus Ehikioya, Debo Eboda, Dare Daramola, Tunde Ibiyemi, David Olatunde-Lamidi, Yinka Fadeyibi, Ike Adigwe. Also for Philip Uwumarogie, Akanimo Udoh and Biodun Tobun who went from colleagues to being such loyal friends. For Oluyomi and Lanre Ayedun for reminding me how to persevere.

Practical, Scalable Algorithms for Byzantine agreement

by

Olumuyiwa Oluwasanmi

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor Of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2011

Practical, Scalable Algorithms for Byzantine agreement

by

Olumuyiwa Oluwasanmi

B.Sc., Computer Science and Mathematics O.A.U., Ile-Ife, 2000

M.S., Computer Science, University of New Mexico, 2008

M.S., Mathematics, University of New Mexico, 2009

PhD, Computer Science, University of New Mexico, 2011

Abstract

With the growth of the Internet, there has been a push toward designing reliable algorithms that scale effectively in terms of latency, bandwidth and other computational resources. Scalability has been a serious problem especially with peer-to-peer (p2p) networks which may have sizes of more than a million nodes.

An important problem in designing reliable algorithms is Byzantine agreement. For reasons of scalability, message complexity is a critical resource for this problem. Unfortunately, previous solutions to Byzantine agreement require each processor to send $O(n)$ messages, where n is the total number of processors in the network.

In this dissertation, we show that the Byzantine agreement problem can be solved with significantly less that a linear number of messages both in theory and in practice. We implement and test algorithms that solve the classical problem with each processor sending only $\tilde{O}(\sqrt{n})$ messages. Further, we consider the problem in the case where we assume the existence of a random beacon: a global source of random bits. We show that with this assumption, the required number of messages drops to $O(\log n)$, with small hidden constants.

Our algorithms are Monte Carlo and succeed with high probability, that is probability $1 - o(n^k)$ for some positive constant k. Our empirical results suggest that our algorithms may outperform classical solutions to Byzantine agreement for network of size larger than 30,000 nodes.

Contents

List of Figures **xv**

Glossary **xvii**

1 Introduction **1**

1.1 Introduction and motivation 1

1.1.1 Motivation 2

1.2 Byzantine agreement problem formal description 3

1.2.1 Modeling faults: 4

1.3 Main Results 6

1.3.1 Dissertation organization 8

2 Related work and technical preliminaries **9**

2.1 Applications of Byzantine agreement 9

2.1.1 Byzantine Fault-Tolerant Systems 9

2.1.2 State machine replication 10

2.1.3 Secure multiparty computation 10

2.2 Lower bounds 11

2.3 Probabilistic protocols 11

2.4 Almost-everywhere Byzantine agreement 12

2.5 Algorithmic Techniques 14

2.5.1 Samplers 14

2.6 Universe Reduction 15

2.6.1 Committee elections, Election graph and Feige's algorithm 16

2.7 Random Beacon, Secret Sharing and Byzantine Agreement and Measuring latency 18

2.7.1 Random Beacon assumption 18

2.7.2 Secret Sharing 19

2.7.3 CKS algorithm 19

2.7.4 Measuring latency 20

3 Non-adaptive adversary **21**

3.1 Introduction 21

3.1.1 Model 22

3.1.2 Problems 23

3.1.3 Overview of Results 23

3.1.4 Related Work 24

3.2 Our Algorithm 25

3.2.1 Almost-Everywhere Universe Reduction and BA . . . 26
3.2.2 Almost-Everywhere to Everywhere . . . 28
3.3 DETAILED DESCRIPTION OF OUR ALGORITHM AND THE ELECTION GRAPH. . . 32
3.3.1 COMMITTEES . . . 32
3.3.2 COMMITTEE SELECTION . . . 33
3.3.3 ELECTION DESCRIPTION . . . 33
3.3.4 ALMOST-EVERYWHERE BYZANTINE AGREEMENT PROTOCOL . . . 36
3.3.5 COMMUNICATION TREE AND MONITORING SETS . . . 36
3.3.6 DETAILED PROOFS . . . 39
3.4 EXPERIMENTS . . . 42
3.4.1 EXPERIMENTAL SETUP . . . 42
3.4.2 Experimental Results . . . 43

4 Adaptive Adversary **48**
4.1 Introduction . . . 48
4.1.1 Our Contributions . . . 48
4.1.2 Chapter Organization . . . 49
4.2 SHALLOWTREE Algorithm . . . 49
4.2.1 Adversarial Model . . . 49
4.2.2 Election Tree Description . . . 50
4.2.3 High-level algorithm description . . . 51

4.3 DEEPTREE Algorithm 52

4.3.1 Model Description 52

4.3.2 DEEPTREE description 52

4.3.3 Almost-Everywhere Byzantine agreement 53

4.4 Analysis and Proofs 54

4.4.1 SHALLOWTREE 54

4.5 Experiments 58

4.5.1 Setup 60

4.5.2 Results 61

4.6 DEEPTREE algorithm details 62

4.6.1 Algorithmic Description 62

4.6.2 Secret Sharing 63

4.6.3 Samplers 64

4.6.4 Network structure 65

4.6.5 Communication protocols 66

4.6.6 Correctness of communications 67

5 Byzantine agreement with a Random Beacon 69

5.1 Introduction 69

5.2 Algorithms and their Description 70

5.2.1 RBQUERY 70

5.2.2 RBSAMPLER 73

5.3 Analysis and proofs 75

5.3.1 Analysis and proofs for RBQUERY 75

5.3.2 Analysis and proofs for RBSAMPLER 80

5.4 Experimental Results 84

5.4.1 Experimental setup 86

5.4.2 Results 87

6 Conclusion and Future Work **89**

6.1 Conclusions 89

6.2 Future work Byzantine agreement in general 91

6.2.1 Improving Efficiency 91

6.2.2 Lower bounds 92

6.2.3 Asynchronous algorithm 92

6.2.4 Distributed Deployment 92

References **93**

A Algorithms and further technical proofs **100**

A.1 Algorithms 100

A.1.1 Arrays and the election subroutine 100

A.1.2 Algorithm and proofs for ALMOST-EVERYWHERE-BYZANTINE-AGREEMENT-WITH-RANDOM-SOURCE 102

A.1.3 Main protocol for a.e. BA 103

A.1.4 Almost-everywhere to everywhere algorithm 103

List of Figures

2.1 Sampler 15

2.2 Election graph. Note that their are expander graph connections between successive layers of the election graph 16

2.3 Feige's algorithm: Each processor randomly picks a bin; the lightest bin has a fraction of good processors similar to that of the whole population. The red circles are bad processors, the blue circles are the good processors, the numbered cylinders are the bins. 17

3.1 Election graph showing samplers. 29

3.2 Steps 6-10 of almost-everywhere to everywhere Protocol 30

3.3 Algorithm for informing sending messages to monitoring sets. 37

3.4 Algorithm for learning the identity of the processors its committee. 37

3.5 Algorithm for electing the next layer of processors. 38

3.6 Algorithm for electing a set of $\ln^3 n$ processors 38

3.7 Algorithm for Byzantine agreement. 39

3.8 Top: Log of number of nodes vs. average number of messages; Bottom: Log of number of nodes vs log of average number of messages 45

3.9 Top: Log of number of nodes vs. average number of bits sent; Bottom: Log of number of nodes vs log of average number of bits sent 46

3.10 Top: Proportion of bandwidth used by the almost-everywhere part of our algorithm. Bottom: Latency vs. the logarithm of the number of nodes 47

4.1 SHALLOWTREE election tree 50

4.2 Top: Log of number of nodes vs. number of messages sent; Bottom: Log of number of nodes vs max messages sent by a node. 58

4.3 Top: Log of number of nodes vs. number of bits sent; Bottom: Log of number of nodes vs max bits sent by a node. 59

4.4 Algorithm Latency 60

5.1 Top: Log of number of nodes vs. number of messages sent; Bottom: Log of number of nodes vs max messages sent by a node. 84

5.2 Top: Log of number of nodes vs. number of bits sent; Bottom: Log of number of nodes vs max bits sent by a node. 85

5.3 Algorithm Latency 86

6.1 Total bits sent by all algorithms we described in this dissertation 90

Glossary

w.h.p.(With high probability)	With high probability means this event occurs with probability $1 - 1/n^c$, where c is a positive constant and n is the number of nodes in the network.
polylog	Poly-logarithmic implying some logarithmic polynomial effectively, $O(\log^k n)$, where k is some positive constant.
$\tilde{O}$	The order function multiplied by some polylog n. For example $\tilde{O}(n^2)$ is $O(n^2 \log^k n)$. Where k is some positive constant.

Chapter 1

Introduction

1.1 Introduction and motivation

Failure is a fact of life. As computer networks grow larger, there is a need to build services with protocols that are fault tolerant. One important type of fault tolerance is tolerance to the failures of nodes in a network. In this dissertation, we focus on designing protocols robust to this type of fault. In particular we focus on the Byzantine agreement problem.

Byzantine agreement can be defined as follows. There are n processors some of which are good (follow the protocol) and some of which are bad (deviate in an arbitrary way). The goal is to ensure all good processors output the same bit and that this bit equals the input bit of some good processor.

1.1.1 Motivation

Byzantine agreement is an important problem because it shows how to build a network that is more reliable than its individual components. One simple way of using Byzantine agreement (BA) in distributed computation is for each processor of the network to vote on the outcome of the computation, send its vote to every other processor, and then have all processors perform Byzantine agreement to decide what the output should be.

Given the above, it is not surprising that BA is used in areas as diverse as: trustworthy computing, in systems like PBFT [19, 21], SINTRA [67, 15], Zyzzyva [46], and Aardvardk [24, 25]; peer-to-peer networks, in systems such as Oceanstore [47] and Farsite [2]; databases, in distributed commit protocols [56, 73]; and control systems, for example, the SIFT system for flying aircraft [71, 55]. Unfortunately, current BA algorithms have a problem:

- *"Eventually batching cannot compensate for the quadratic number of messages [of Practical Byzantine Fault Tolerance (PBFT)]"* [26]
- *"The communication overhead of Byzantine Agreement is inherently large"* [22]
- *"Unfortunately, Byzantine agreement requires a number of messages quadratic in the number of participants, so it is infeasible for use in synchronizing a large number of replicas"* [64]

Since Byzantine agreement is an important tool for building fault-tolerant systems, it is crucial that the algorithms that solve it are scalable in the sense that their resource costs do not increase significantly as the system increases in size. The goal of scalability is the main focus of this dissertation.

1.2 Byzantine agreement problem formal description

The Byzantine agreement problem was first formulated in a paper by Lamport, Shostack and Pease in [49] to model fault tolerance in computing systems.

We now describe the problem by a motivating scenario used by Lamport et al. [49] and Micali and Feldman in [34]. A General is giving orders on the night before a great battle. There are a set of lieutenants, some loyal and some traitorous. Each lieutenant receives a message describing the battle strategy for the next day. The message consists of a single word: retreat or attack. If all good lieutenants attack, the army will defeat the enemy, beating them into a hasty retreat. If all good lieutenants retreat, the army will survive to organize another battle campaign at some other date. However, if some good lieutenants retreat and some attack, the army would be completely obliterated. The goal is to :

1. Ensure all good lieutenants commit to a common course of action.
2. Ensure that in the case the General gives the same order to each lieutenant, that all lieutenants commit to that order.

These goals may not be achieved because of a traitorous General sending different messages to the lieutenants or traitorous lieutenants misleading their peers (traitorous lieutenants can give different opinions to the good ones, or refuse to communicate their orders to other lieutenants). Defining the problem more formally we have the following assumptions:

- There exists a set of n processors with a certain number good, and the remaining bad.
- Bad processors are controlled by an adversary who can coordinate their actions. The bad processors may behave in any arbitrary way, or send any type of message they desire.
- Each processor's identity is unique and known to everyone.

- All processors can send messages to one another. We assume point-to-point communication i.e. any processor can send a message directly to any other processor and the recipient of the message can verify the sender's identity.
- The communication network can be either:
  - *Synchronous*: There is an upper bound on the time it takes to deliver a message and this upper bound is known to all processors.
  - *Asynchronous*: There is no upper bound on how long it takes to send a message.

Each processor, at the beginning of the protocol starts with an initial input bit. Our goal is at the end for each processor i to output a bit b_i such that the following properties hold:

- *Agreement* : All good processors should have the same bit value at the end:$\forall i, j, b_i = b_j$.
- *Non triviality*: If all processors have the same input bit then that bit is the output bit for all good processors.
- *Termination*: The protocol must terminate.

Synchronous Byzantine agreement algorithms proceed in *rounds*: These are fixed units of time in which the sending and reception of messages is guaranteed. Asynchronous algorithms however refer to a step of the algorithm in which each processor sends or receives messages. There is no guarantee that messages will be delivered. One important property of Byzantine agreement algorithms is the number of rounds it takes to terminate (latency). Another important property is the number of messages and bits sent.

1.2.1 Modeling faults:

We now describe two important types of node faults:

- *Fail-stop faults*: These occur when a processor completely stops, no longer sending out any messages.
- *Byzantine faults*: These occur when a processor arbitrarily deviates from the steps specified by the algorithm. This behavior can be malicious with the intent of disrupting the successful execution of the algorithm. It may also involve coordination between processors suffering faults. We say processors that suffer Byzantine faults are controlled by an adversary.

There is also a need to define the characteristics of the adversary controlling the bad processors in the case of Byzantine faults. Some of these characteristics include:

- Computational power:
  - *Computationally unbounded*: The adversary can perform any computation instantly, even problems taking more than polynomial time.
  - *Computationally bounded*: The adversary is limited in computational resources, that is the adversary can only solve problems taking polynomial time. That implies that the adversary cannot break cryptographic protocols, assuming standard cryptographic assumptions.
- Communication power:
  - *Rushing*: This assumes the adversarial nodes receive all messages sent to them early in the round early and may construct their own messages and decide where to send them, based on information received.
  - *Non-rushing*: All messages leave the senders at the same time.
  - *Private channels*: A message between two good processors cannot be seen by the adversary.
  - *Public channels*: A message between two good processors can be seen by the adversary. This is sometimes referred to as the *full information* model.

- *Fault tolerance*: The threshold number of processors the adversary can take over. This is usually referred in literature as the variable t.

- Corruption Power:
  - *Adaptive*: An adaptive adversary can take over processors at any point in the protocol up taking over t processors and once processors are corrupted, they stay corrupted [51].
  - *Non-adaptive*: A non-adaptive adversary must choose the set of nodes to corrupt before the algorithm begins, this is sometimes also referred to as a *Static* adversary. In some cases the adversary can carefully select the nodes to corrupt after seeing the algorithm. In other cases, the Byzantine nodes are selected randomly according to some distribution.

1.3 Main Results

In this dissertation we describe algorithms for solving Byzantine agreement where each processor sends at most $\tilde{O}(\sqrt{n})$ [1] bits per processor, while keeping latency at most $O(\log n)$.

Our algorithms follow the results [44, 45, 59, 43, 42] that focus on scalability. For some of our algorithms, we do not make any cryptographic assumptions. We consider algorithms tolerating a range of faults from $\sqrt{n}$ to $n/3$ fraction of the network size. All of our algorithms work in the synchronous model of communication and they all assume a rushing adversary. Lastly, we have a class of algorithms that depend on the assumption of the existence of a Random Beacon, which is a stream of random bits known to all processors. We show that with this assumption, we can

[1] $\tilde{O}$ notation means the order function multiplied by some polynomial in $\log n$. For example $\tilde{O}(n^2)$ is $O(n^2 \log^k n)$, where k is some positive constant.

significantly reduce communication and latency costs much further than the previous class of algorithms. Our algorithms are all Monte Carlo and succeed with high probability.

We now present some of the main results which are:

- In Chapter 3 we describe an algorithm (NATREE) robust against a non-adaptive adversary. It assumes private channels and the existence of a bit-commitment and digital signature scheme. It sends $\tilde{O}(\sqrt{n})$ messages per processor with $O(\log n)$ latency. It tolerates $t \leq n\left(\frac{1}{8} - \epsilon\right)$ and assumes private channels. Part of this work was previously published in [59].

- In Chapter 4 we describe algorithms robust against an adaptive adversary. They both assume private channels and send $\tilde{O}(\sqrt{n})$ messages per processor with $O(\log n)$ latency. The first algorithm (SHALLOWTREE), tolerates $t < n(1/6 - \epsilon)$ non-adaptive faults distributed uniformly at random and $\sqrt{n}$ adaptive faults. The second algorithm (DEEPTREE), tolerates $t < n(1/3 - \epsilon)$ adaptive faults.

- In Chapter 5, we describe two algorithms (RBQUERY, RBSAMPLER) that assume the existence of a Random Beacon. They are both robust against an adaptive adversary, work in the full information model, tolerate $t \leq n\left(\frac{1}{3} - \epsilon\right)$ adaptive faults and have $O(1)$ expected latency. The algorithm RBQUERY requires $O(\log n)$ messages sent per processor on average and assumes the bad processors can only send at most $O(\log n)$ messages. The algorithm RBSAMPLER requires each processor to send at most $polylog(n)^2$ messages, no matter how many messages the bad processors can send. However, it only achieves almost-everywhere Byzantine agreement.

$^2O(\log^k)$ for some positive constant k

1.3.1 Dissertation organization

The rest of the dissertation is organized as follows. In Chapter 2 we give a survey of related work, applications and describe algorithmic techniques used in our results. In Chapter 3 we give the details of our algorithm that is robust to a non-adaptive adversary and its experimental results. In Chapter 4 we give the details of our algorithms that are robust to an adaptive adversary and their experimental results. In Chapter 5 we describe our algorithms based using the Random Beacon assumption and give their experimental results. In Chapter 6 we conclude and give details of proposed future work.

Chapter 2

Related work and technical preliminaries

2.1 Applications of Byzantine agreement

We begin this Chapter with a discussion of applications of Byzantine agreement.

2.1.1 Byzantine Fault-Tolerant Systems

Lamport [49] formulated the Byzantine agreement problem as a way of building a multiprocessor fault-tolerant air traffic control system. Castro and Liskov in [21, 20] use the Byzantine agreement protocol to build a Byzantine Fault-tolerant version of NFS called BFS (Byzantine File System). Alvisi et al. [4] use a Byzantine agreement algorithm to build a fault-tolerant data backup service. Molina, Pittelli and Davidson [56] use a Byzantine agreement algorithm to build a fault-tolerant database system. Reiter and Malkhi [52] built a fault-tolerant data repository using a Byzantine agreement algorithm. Agbaria and Friedman use Byzantine agreement to define build intrusion detection systems in [3].

2.1.2 State machine replication

Schneider in [65] defined the concept of state machine replication approach as a general method of implementing fault-tolerant services in distributed systems. He defined a state machine replication method of implementing a fault-tolerant service as *"each service comprises one or more servers and exports operations that clients invoke by making requests. Although using a single, centralized, server is the simplest way to implement a service, the resulting service can only be as fault-tolerant as the processor executing that server. If this level of fault tolerance is unacceptable, then multiple servers that fail independently must be used. Usually, replicas of a single server are executed on separate processors of a distributed system, and protocols are used to coordinate client interactions with these replicas. The physical and electrical isolation of processors in a distributed system ensures that server failures are independent, as required"*. Lamport in [48] described *Paxos* a family of protocols for solving *consensus* and Byzantine agreement. Byzantine agreement in the case when the processors are simply non-responsive is called consensus. Many fault-tolerant systems that use state machine replication have been built using Paxos. Burrows in [14] designed a distributed file locking service on the Google File System using Paxos. Issard in [39] designs a cluster management service based on Paxos called Autopilot. Recently, fault tolerant Key-Value data stores like Keyspace and Megastore [7, 69] have been built based on Paxos.

2.1.3 Secure multiparty computation

Secure multiparty computation is a problem initially formulated by Yao in [72]. One example described by Yao in [72] is the problem where two millionaires want to know who is richer without revealing the true extent of their wealth. More generally there are n processors, each processor i has some private data x_i and they all desire to compute some function $f(x_1, x_2, x_3, \ldots, x_n)$ whose output will be known publicly to all processors without compromising their private data. The problem can be made more challenging by assuming a certain subset of the processors are bad and are trying to prevent computation of f. We observe that secure multiparty computation is very similar to Byzantine agreement, where each x_i is a bit and f is a function that computes the majority bit value. One difference is the requirement in secure multiparty computation that the input bit values remain private. Secure multiparty computation is generally harder than Byzantine agreement, not only because of the privacy constraint on processor's inputs, but also because the function f can be computationally more complicated. Byzantine agreement is frequently used as a subroutine in secure multiparty computation to force processors to send consistent messages. Recent applications of secure multiparty computation are described in [8, 11, 9, 10]

2.2 Lower bounds

It was proved by Lamport, Shostak and Pease in [49] that Byzantine agreement is only possible when less than $n/3$ of the processors are bad. We describe the high-level idea behind this result is describe by Micali and Feldman in [34]. Fischer, Lynch and Paterson in [35] also proved that a single Byzantine fault in the asynchronous model of communication renders Byzantine agreement impossible for a deterministic algorithm.

Dolev in [27] showed that if $\Theta(n)$ processors are bad, $\Omega(n^2)$ messages must be sent to ensure Byzantine agreement with probability 1. Dolev and Reishuck in [28] show that even with the assumption of the existence of a digital signature scheme (computationally bounded adversary), there is a lower bound of $\Omega(n + t^2)$ messages to ensure Byzantine agreement with probability 1.

2.3 Probabilistic protocols

Rabin in [62] solved a randomized version of the Byzantine agreement problem using the notion of a *common coin* (we also refer to this as a Random Beacon). A common coin is a value that is only available to all the processors at the beginning of each round of computation. It is 0 with probability $1/2$ and 1 with probability $1/2$. Rabin's algorithm [62] takes $O(1)$ expected rounds and requires $\Theta(n^2)$ messages. Rabin's algorithm works in the asynchronous, full information model and is correct with probability 1.

Several results allow for solving Byzantine agreement with randomized algorithms even without a global coin. In [41] Karlin and Yao prove a $n - t > 2n/3$ lower bound for randomized Byzantine agreement. Feldman and Micali [34] give a randomized algorithm that has $O(1)$ expected latency and tolerates $t < n/3$ adversarial faults. Their algorithm works in the synchronous model of communication with private channel and makes standard cryptographic assumptions. Their algorithm is correct with probability 1. Recently, King et al. in [44, 38, 45, 59, 42, 43] show that we can solve Byzantine agreement with high probability sending $\tilde{O}(\sqrt{n})$ messages and with $O(\log n)$ latency in both the synchronous and asynchronous model of communication. Some of their results require private channels, but they always assume a computationally unbounded adversary.

2.4 Almost-everywhere Byzantine agreement

In [29] Dwork et al. defined the concept of *almost-everywhere Byzantine agreement* as follows. A protocol achieves almost-everywhere agreement if the fraction of the good processors that decide the correct bit approaches 1 as $n \to \infty$. They give an algorithm for solving almost-everywhere Byzantine agreement even when the communication graph between the processors has maximum degree $O(1)$. The communication model of their algorithm is synchronous tolerates a computationally unbounded adversary. However, their algorithm only tolerates $O(n/\log n)$ Byzantine faults.

Upfal in [70] solved almost-everywhere Byzantine agreement in the same model with a bounded degree communication graph. His algorithm can tolerate $\Theta(n)$ Byzantine faults, but like [29], always requires $\Omega(n^2)$ messages.

	Adapt/ NonAd	Full Info/ Cryp/Priv	Sync/ Async	Fract. in Agreemt	Error Prob	Bits/ Proc	Time	Resilience
[44]	NonAd	Full Info	Sync	$1-\frac{1}{\log n}$	$1/n^c$	$\log^c n$	$\log^c n$	$\frac{n}{3+\epsilon}$
[36]	NonAd	Full Info	Sync	1	0	n^c	$\log n$	$\frac{n}{3+\epsilon}$
[44, 42]	NonAd	Full Info	Sync	1	$1/n^c$	$\tilde{O}(n^{1/2})$	$\log^c n$	$\frac{n}{3+\epsilon}$
[43]	Adapt	Priv	Sync	$1-\frac{1}{\log n}$	$1/n^c$	$\log^c n$	$\log^c n$	$\frac{n}{3+\epsilon}$
[43]	Adapt	Priv	Sync	1	$1/n^c$	$\tilde{O}(n^{1/2})$	$\log^c n$	$\frac{n}{3+\epsilon}$
NATREE	NonAd	Priv	Sync	1	$1/n^c$	$\tilde{O}(n^{1/2})$	$\log^c n$	$\frac{n}{6+\epsilon}$
SHALLOWTREE$^+$	Adapt	Priv	Sync	1	$1/n^c$	$\tilde{O}(n^{1/2})$	$\log^c n$	$\sqrt{n}$
DEEPTREE	Adapt	Priv	Sync	1	$1/n^c$	$\tilde{O}(n^{1/2})$	$\log^c n$	$\frac{n}{3+\epsilon}$
RBQUERY†	Adapt	Full Info	Sync	1	$1/n^c$	$\tilde{O}(1)$	$1E$	$\frac{n}{3+\epsilon}$
RBSAMPLER†	Adapt	Full Info	Sync	1	$1/n^c$	$\tilde{O}(1)$	$1E$	$\frac{n}{3+\epsilon}$
[13]	Adapt	Full Info	Async	1	0	$n2^n$ E	2^n E	$n/3-1$
[40]	NonAd	Full Info	Async	1	$1/\log^c n$	$\tilde{O}(n)$	$\log^c n$	$\frac{n}{3+\epsilon}$
[16]	NonAd	Cryp	Async	1	$1/n^c$	$\tilde{O}(n)$ E	1 E	$n/3-1$
[61]	Adapt	Priv	Async	1	0	$\tilde{O}(n^2)$ E	1 E	$n/4-1^*$
[68]	Adapt	Cryp	Async	1	$1/n^c$	$\tilde{O}(n^2)$ E	1 E	$n/3-1$

Table 2.1: *State of the art on relevant BA algorithms.* If E then expected number, else worst case; values are asymptotic; $\tilde{O}$ indicates extra $\log$ factors; c, ϵ *any* positive constants. * $n/3-1$ resilience is achievable with n^c E time [1] or constant time and error > 0 [18, 60]. Items in boldface refer to results in this dissertation and their respective chapters. † Assumes Random Beacon. $^+$ Can also tolerate $\frac{n}{6+\epsilon}$ Byzantine faults distributed uniformly at random.

In Table 2.1, we summarize state of the art results for Byzantine agreement. Results from this dissertation are in boldface.

2.5 Algorithmic Techniques

We now describe some of the algorithmic techniques frequently used in this dissertation.

2.5.1 Samplers

Our protocols makes use of samplers, which are a special family of bipartite graphs that define subsets of elements such that all but a small number of the selected subsets contain close to the fraction of bad elements in the whole set. We use the definition of samplers found in [40] which is equivalent to the one defined in [74].

Definition 2.5.1. *Let $[r]$ denote the set of integers $\{1 \ldots r\}$, and $[s]^d$ the multisets of size d consisting of elements of $[s]$. Let $H : [r] \rightarrow [s]^d$ be a function assigning multisets of size d to integers. We define the size of the intersection of a multiset A and a set B to be the number of elements of A which are in B. Then we say H is a (θ, δ) sampler if for every set $S \subset [s]$ at most a δ fraction of all inputs x have $\frac{|H(x) \cap S|}{d} > \frac{|S|}{s} + \theta$.*

The following Lemmata establishing the existence of samplers can be shown using the probabilistic method. For $s' \in [s]$, let $deg(s') = \{|r' \in [r]|s : t : s \in H(r')\}$ [1]. A slight modification of Lemma 2 in [74] yields:

Lemma 2.5.1. *For every $r, s, d, \theta, \delta > 0$ such that $2 \log_2(e) d\theta^2\delta > s/r + 1 - \delta$, there exists a (θ, δ) sampler $H : [r] \rightarrow [s]^d$ and for all $s \in [s]$, $deg(s) = O((rd/s) \log n)$.*

Lemma 2.5.2. *For every $s, \theta, \epsilon \geq 0$ and $r \geq s/\epsilon$, there exists a constant c such that for all $d \geq c \log(1/\epsilon)/\theta^2$ there is a (θ, ϵ) sampler $H : [r] \rightarrow [s]^d$.*

[1] $deg(\ldots)$ refers to the degree or the number of neighbors of a node in a graph.

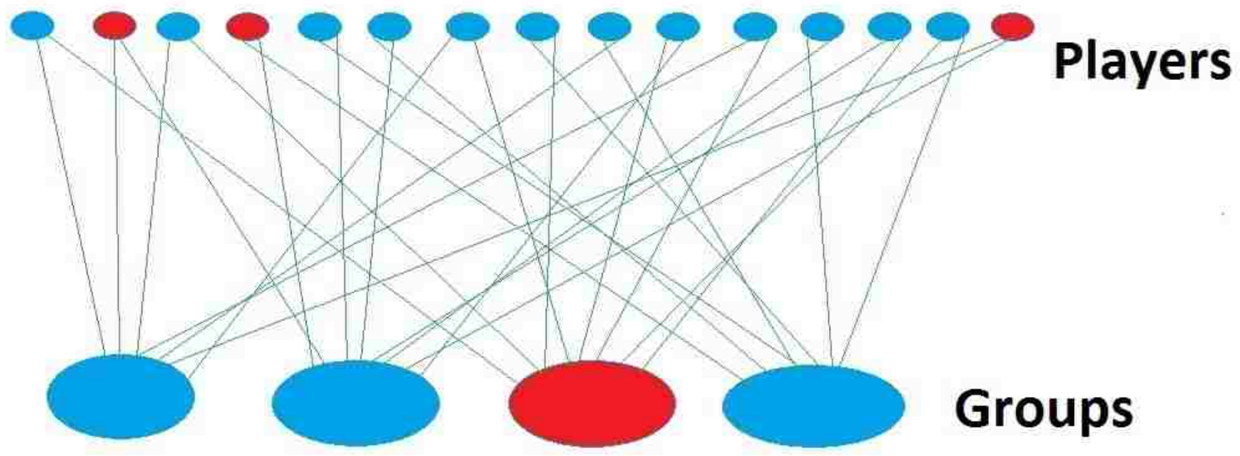


Figure 2.1: Sampler

2.6 Universe Reduction

Gradwhol et al. in [37] defined the concept of *Universe reduction*. The Universe reduction problem as described by King and Saia in [42] *"is to bring processors to agreement on a small subset of processors with a fraction of bad processors close to the fraction for the whole set. I.e., the protocol terminates and each good processor outputs the same set of processor ID's such that this property holds"*.

In our algorithms we frequently use Universe reduction as a subroutine. We run Byzantine agreement among the small subset of processors selected and have them communicate their output to the remaining processors. We describe this approach in more detail in the next section.

2.6.1 Committee elections, Election graph and Feige's algorithm

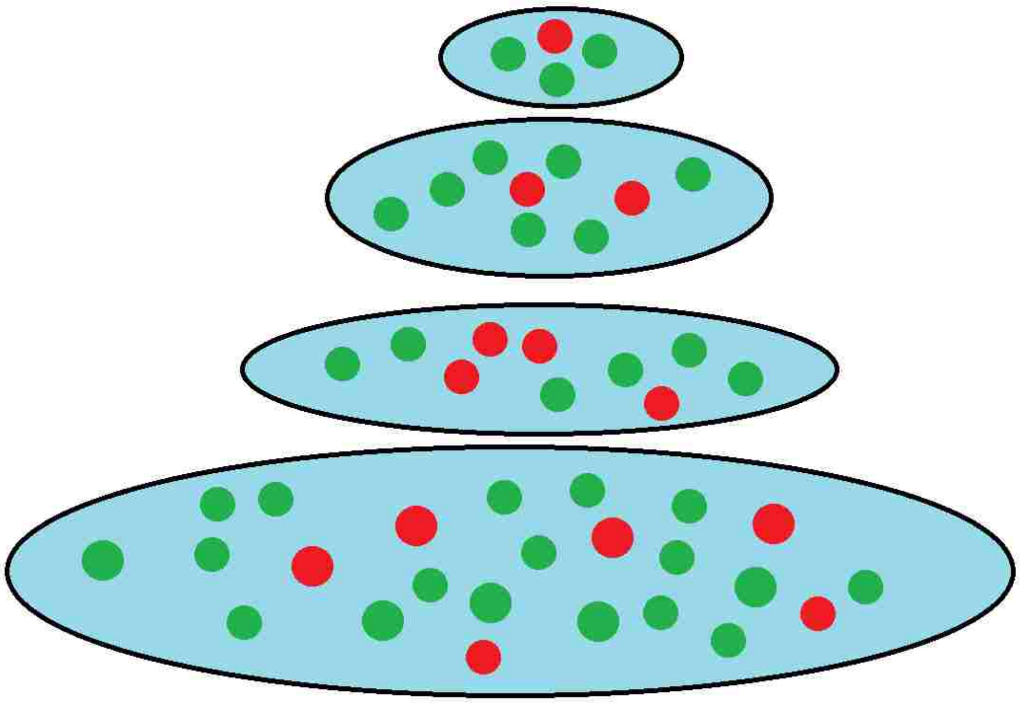


Figure 2.2: Election graph. Note that their are expander graph connections between successive layers of the election graph

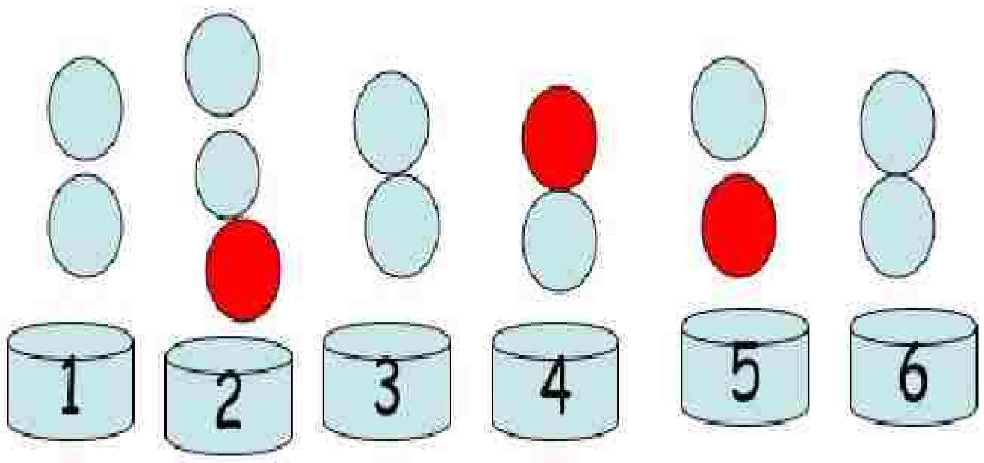


Figure 2.3: Feige's algorithm: Each processor randomly picks a bin; the lightest bin has a fraction of good processors similar to that of the whole population. The red circles are bad processors, the blue circles are the good processors, the numbered cylinders are the bins.

Our algorithms follow closely work done by King et al. in [44, 45, 40, 59]. In these papers, the algorithms are scalable in the sense that each good processor sends out $o(n)$ messages. A tool we borrow from these papers is an election graph.

In an election graph Figure 2.2, each node represents a committee, a small subset of the processors that will hold an election amongst themselves. The elections progress in steps from the bottom layer of the election graph to the top. The processors elected at layer i participate in the elections at layer $i + 1$. The mapping between the winners at layer i to the committees at layer $i + 1$ is given by a sampler. All processors participate in the committees at the bottom layer, but the number of processors remaining decreases geometrically until there is a very small number of processors at the very top layer.

We make use of Feige's algorithm [33] to perform elections at each node in the Election graph. In Feige's algorithm Figure 2.3, there are a number of bins and initially, each good processor selects a bin uniformly at random. After processors choose their bins, the processors in the bin with the smallest number of processors are elected. The intuition here is that the adversary can not increase the fraction of bad processors in the lightest bin by that much over the fraction of bad processors in the node.

Frequently, we will use the Election graph to compute Universe reduction. Once we have performed Universe reduction, the small, representative committee elected can use any heavy-weight algorithm to perform Byzantine agreement amongst themselves. They can then send their outputs to the rest of the processors using the communication infrastructure of the election graph.

2.7 Random Beacon, Secret Sharing and Byzantine Agreement and Measuring latency

2.7.1 Random Beacon assumption

A Random Beacon is a random stream of bits known to all the processors. Rabin, Maurer et al. [63, 53, 17, 5, 30] describe a computation model called the Memory Limited or Limited Storage space model, which assumes the existence of a Random Beacon. They use this assumption to design an efficient, provably secure cryptographic protocol. Lee, Chang and Chan show in [50] how to build a Random Beacon using publicly available information on the Internet. Clark and Hengartner in[23] show how to build a Random Beacon using data available from financial markets such as the Dow Jones Industrial Average (DJIA) or other financial data. Eastlake in [31, 32] also describes how to build a selection protocol from a Random Beacon constructed from random information publicly available on the Internet. In Chapter 5, using the assumption of a Random Beacon, we build more efficient Byzantine agreement algorithms.

2.7.2 Secret Sharing

Another technique we frequently use is secret sharing. Secret sharing [66] is a scheme to allow a dealer to divide a piece of information (a secret) among a group of participants (players). Each player initially has a part of the information called a share and the secret can only be reconstructed from a certain number of shares. An (n, t) secret sharing scheme is an algorithm by which a dealer distributes a secret among n players, and the secret can only be reconstructed from the shares held by at least t players. An example of an (n, t) secret sharing protocol is the one specified by Shamir in [66]. In this scheme a random polynomial (coefficients picked uniformly at random) of degree $t - 1$ is chosen with the constant coefficient (polynomial evaluated at 0) being the secret. The dealer chooses n different points on the the polynomial curve uniformly at random, and each player receives a distinct point. This scheme works since t interpolation points are necessary and sufficient to allow full recovery of the coefficients of the polynomial of degree t.

2.7.3 CKS algorithm

One of the most efficient Byzantine agreement algorithms is due to Cachin, Kursawe and Shoup [16]. In experiments in this dissertation we frequently compare our algorithms with this algorithm, which we now describe in more detail. The algorithm tolerates $t < n/3$ adversarial faults, with a computationally bounded, rushing adversary with asynchronous communication and private channels. The algorithm sends $O(n^2)$ messages in total and terminates in $O(1)$ rounds in expectation. The algorithm assumes the existence of a trusted dealer to distribute cryptographic keys. A distributed global coin tossing scheme is used to break ties when the votes are close. The global coin tossing scheme strongly relies on cryptographic assumptions such as digital signatures and the decision and computational Diffie-Hellman assumptions. We make use of this algorithm as a subroutine in our Byzantine agreement algorithm described in Chapter 3.

2.7.4 Measuring latency

In the experimental results section of chapters 3, 4 and 5 we measure algorithmic latency in experiments. We make the assumption that the latencies of our algorithms are dominated by the time it takes to send a message and we can use this as the unit cost of an operation. We cite the following paper by Bhatele and Kale [12] which provides information via benchmarks about what these numbers might mean in real world applications. In [12] the values for message latencies are between 5 and 16 milliseconds for the largest message size in the paper (1MB). The benchmarks in the paper consider models that assume the existence or lack thereof, of resource contention. Throughout this dissertation, we use this assumption in our measurement of the latencies of our algorithms.

Chapter 3

Non-adaptive adversary

3.1 Introduction

In this chapter, we design, implement and simulate an algorithm that solves Byzantine agreement against a non-adaptive adversary with each processor sending $\tilde{O}(\sqrt{n})$ bits per processor. We assume that bad processors can send *any* number of messages. The contributions of this chapter are as follows:

- We design a new algorithm which is based on, but more practical than, the algorithm from [42]. Our new algorithm significantly reduces the constants compared to the previous algorithm through use of cryptography.
- We implement and simulate our new algorithm, showing empirically that for large networks, it can achieve consensus with significantly less bandwidth than algorithms that are currently used in practice.

While our algorithm in this chapter has resource costs that are asymptotically no better than the result of [42] (with respect to $\tilde{O}()$ notation), in practice, the algorithm is significantly more efficient.

3.1.1 Model

We assume a fully connected network of n processors, whose IDs are common knowledge. Each processor has a private coin. Communication channels are authenticated, in the sense that whenever a processor sends a message directly to another, the identity of the sender is known to the recipient. We assume a *non-adaptive* (sometimes called *static*) adversary. That is, the adversary chooses the set of t bad processors at the start of the protocol, where t is a constant fraction, of the number of processors n.

We assume a partially synchronous communication model: any message sent from one honest player to another honest player needs at most Δ time steps to be received and processed by the recipient for some fixed Δ, known to all players. We assume that the clock speeds of the honest players are roughly the same. Our algorithm makes use of a distributed random number generating algorithm from [6] and the Universe reduction algorithm from [16] as a subroutine. Thus, we must make the same assumptions as in those papers. Namely, we assume the existence of :

1. public key cryptography (but *not* a public key infrastructure[1])
2. a digital signature scheme; and
3. a bit commitment scheme h, i.e. a function such that $h(x)$ reveals nothing about x.

We overcome the lower bound of [28] by allowing for a small probability of error. In particular, the $\Omega(n^2)$ lower bound on the number of messages to compute Byzantine agreement deterministically implies that any randomized protocol which always uses $o(n^2)$ messages must err with some probability $\rho > 0$, since with probability $\rho > 0$, an adversary can guess the an adversary can guess

[1]A public key infrastructure or PKI, is a system for the management of and verification of identities, digital certificates, user roles, hardware and software. It also involves establishing a Certificate Authority. This inherently not scalable to set up, as this requires for every processor to have the public key of every other processor.

a sequence of coin flips, and cause the protocol to fail if those coin-flips occur. As the algorithm becomes deterministic from the point of view of the adversary. Thus, any randomized algorithm that always achieves $o(n^2)$ messages must necessarily be a Monte Carlo algorithm.

3.1.2 Problems

We consider two problems in addition to Byzantine agreement. The *leader election* problem is the problem of all processors agreeing on a good processor [44]. The *Universe reduction* problem [37] is to bring processors to agreement on a small subset of processors with a fraction of bad processors close to the fraction for the whole set. That is, the protocol terminates and each good processor outputs the same set of processor ID's such that this property holds. For each of these problems, we say the protocol solves the problem with probability ρ if, given any worst case adversary behavior, including choice of initial inputs, the probability of success of any execution over the distribution of private random coin tosses is at least ρ.

Almost-everywhere Byzantine agreement, Universe reduction, and leader election is the modified version of each problem where instead of bringing all good processors to agreement, a large majority of, but not necessarily all, good processors are brought to agreement.

3.1.3 Overview of Results

We show that by making use of the cryptographic assumptions detailed above and by relaxing the fraction of bad processors to $1/8$, we can significantly improve the communication costs and improve the load balancing characteristics while keeping the latency the same or making it slightly better. Our algorithm works in the synchronous model of communication, although we conjecture that our algorithm could be converted to make it asynchronous. Our research is an extension of two previous papers [44, 45] which introduce the concept of an election graph with groups of processors called committees. The algorithms in these two papers use Feige's protocol described in [33] to elect processors in committees in successive layers of the election graph. The use of Feige's protocol in the previous algorithms carry a heavy penalty in terms of the message complexity for any reasonable size of networks.

Our new algorithm has two parts. The first part is an almost-everywhere Byzantine agreement algorithm similar to [44, 45] except that instead of using Feige's protocol we use a protocol by Awerbuch and Scheideler [6] to elect processors in the committees. This allows us to make the sizes of the committees much smaller, which leads to a significant improvement on the message complexity performance of our algorithms from [44, 45]. Secondly, we implement and make use of a protocol recently described in [42] which allows us to go from almost-everywhere Byzantine agreement to everywhere Byzantine agreement using only $\tilde{O}(\sqrt{n})$ messages. In particular, this new algorithm ensures that with high probability all good processors in our new algorithm learn the correct bit, unlike the algorithms in [44, 45].

3.1.4 Related Work

The algorithm presented in this chapter, along with the algorithm of [42], use randomization to break through the 1985 $\Omega(n^2)$ barrier [28] for message and bit complexity for Byzantine agreement in the deterministic synchronous model, if we assume the adversary's choice of bad processors is made at the start of the protocol, i.e., independent of processors' private coin flips. As mentioned above, the algorithm in this paper also makes use of algorithms from [44, 45] to solve the almost-everywhere Byzantine agreement problem.

In the empirical section of this chapter, we compare the resource costs of our algorithm with the Byzantine agreement algorithm proposed by Cachin, Kursawe and Shoup [16]. Their algorithm withstands up to $n/3$ bad processors, runs in constant expected time, and sends $\Theta(n^2)$ messages. However, unlike our algorithm, their algorithm requires a trusted dealer to distribute cryptographic keys initially in order to set up a public key infrastructure. We emphasize that our algorithm does *not* require the establishment of a public key infrastructure. As pointed out in section 3.1.1, the algorithm we describe in this paper is partially synchronous, while the algorithm of Cachin, Kursawe and Shoup is asynchronous.

Organization of the chapter In Section 3.2 we describe our algorithm. In Section 3.4 we empirically evaluate our algorithm. In section 3.3 we describe the algorithmic details and proofs. Finally, in Section 6.1 we conclude and describe open problems.

3.2 Our Algorithm

Our algorithm consists of two parts:

1. A procedure for solving almost-everywhere Universe reduction (and BA);
2. A procedure for going from almost-everywhere Universe reduction to everywhere BA.

Our procedure for solving almost-everywhere Universe reduction is essentially the same as that of [44], with the following two differences: (1) we replace the committee election protocol used in [44], with a new election protocol based on a random number generation protocol by Awerbuch and Scheideler [6]; and (2) we reduce the size of all committees from $O(\log^3 n)$ to $O(\log n)$ and make the appropriate changes in our election graph. This reduction in committee size is made possible by the new election protocol from [6], which makes use of a cryptographic commitment scheme. It is this reduction in committee size that leads to significant savings in bandwidth over the protocol of [44]. We note that we are not particularly dependent on the algorithm from [6]; any robust, distributed random number generating algorithm will work. We simply use this algorithm because it is the current state of the art in terms of resource costs.

Below, we sketch our entire protocol, details are given in section 3.3.

3.2.1 Almost-Everywhere Universe Reduction and BA

We first describe our protocol to compute almost-everywhere Universe reduction, based on [44]. The processors are assigned to groups of logarithmic size; each processor is assigned to multiple groups. In parallel, each group then elects a small number of processors from within their group to move on. We then recursively repeat this step on the set of elected processors until the number of processors left is logarithmic. Although this approach is intuitively simple, there are several complications that must be addressed.

(1) The groups must be determined in such a way that the election mechanism cannot be sabotaged by the bad processors.
(2) After each step, each elected processor must determine the identities of certain other elected processors, in order to hold the next election.
(3) Election results must be communicated to the processors.
(4) To ensure load balancing, a processor which wins too many elections in one round cannot be allowed to participate in too many groups in the next round.

We address these problems as follows. Item (1): we use a layered network with extractor-like properties. Every processor is assigned to a specific set of nodes on layer 0 of the network. In order to assign processors to a node A on layer $\ell > 0$, the set of processors assigned to nodes on layer $\ell - 1$ that are connected to A hold an election. In other words, the topology of the network determines how the processors are assigned to groups. By choosing the network to have certain desired properties, we can ensure that the election mechanism is robust against malicious adversaries.

To accomplish item (2), we use *monitoring sets*. Each node A of the layered network is assigned a set of nodes from layer 0, which we denote $m(A)$. The job of the processors from $m(A)$ is simply to know which processors are assigned to node A. Since the processors of $m(A)$ are fixed in advance and known to all processors, any processor that needs to know which processors are assigned to A can simply ask the processors from $m(A)$. (In fact, the querying processor only needs to randomly select a polylogarithmic subset of processors from $m(A)$ in order to learn the identities of the processors in A with high probability. This random sampling will be used to ensure load balancing.)

Since the number of processors that need to know the identities of processors in node A is polylogarithmic, the processors of $m(A)$ will not need to send too many messages, but they need to know which processors need to know so they do not respond to too many bad processors' queries. Hence the monitoring sets need to inform relevant other monitoring sets of this information.

Item (3): We use a *communication tree* connecting monitoring sets of children in the layered networks with monitoring sets of parents to inform the monitoring sets which processors won each of their respective elections and otherwise pass information to and from the individual processors on layer 0. Item (4) is addressed by having such processors refrain from further participation.

The protocol results in almost-everywhere agreement on the subset of nodes rather than everywhere agreement, because the adversary can control a small fraction of the monitoring sets by corrupting their nodes. Thus communication paths to some of the nodes are controlled by the adversary. We further note that with this protocol, it is trivial to communicate a bit to almost all of the nodes in addition to communicating a small subset. Thus, it solves both almost-everywhere

BA and almost-everywhere Universe reduction. [2]

3.2.2 Almost-Everywhere to Everywhere

In this section, we describe the Almost-Everywhere to Everywhere protocol of [42]. This is exactly the same protocol as from [42] but we include a description of it here for completeness. In the almost-everywhere protocol sketched above all but a $1/\ln n$ fraction of the good processors agree on a small subset of representative processors (and a bit). This result is proven in [42]. Our goal in this section is to improve the fraction of good processors that agree on the bit. The protocol assumes all processors have used the almost-everywhere agreement protocol to ensure that the following precondition holds:

Precondition: We assume there is a subset C of $O(\log^3 n)$ processors, a majority of which are good; and a bit b that is the input bit of at least one good processor. Each processor p starts with an hypothesis of the membership of C, C_p; this hypothesis may or may not be equal to C or may be empty; and each processor p starts with a value b_p that may or may not be equal to b. The following assumption is critical: there is a set S of at least $(1/2 + \epsilon)n$ good processors, such that for all $p \in S$, $C_p = C$ and $b_p = b$.

Overview of Algorithm: The main idea of this protocol is for each processor p to randomly select $c \log n$ processors to poll as to the membership of C. Unfortunately, if these requests are made directly from p, the adversary can flood the network with “fake” requests so that the good processors are forced to send too many responses. Thus, the polling request are made through the set C, which counts the messages received from each processor to enforce that total number of polling requests sent out is not too large.

[2]Moreover the processors that correctly learn the subset of nodes will also learn the correct bit for BA.

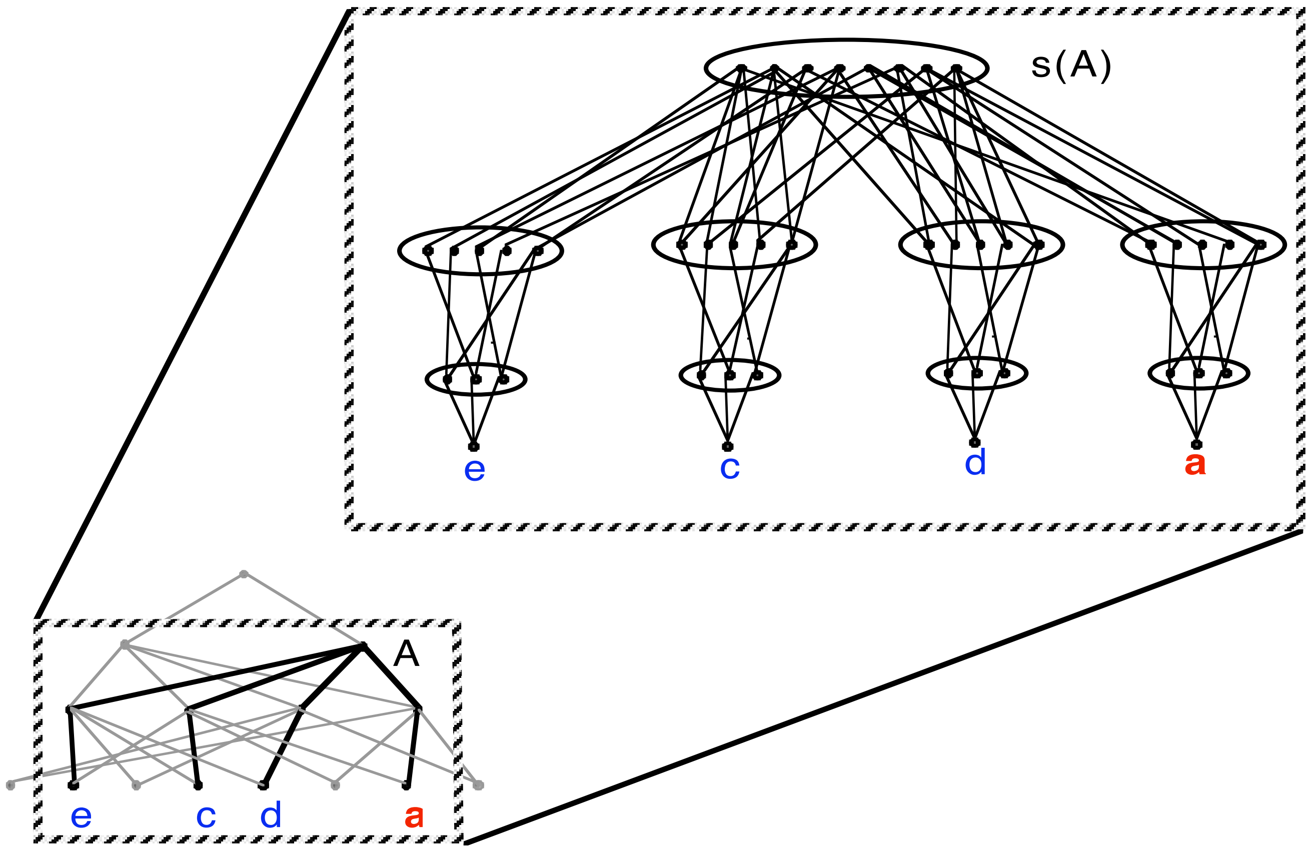


Figure 3.1: Election graph showing samplers.

Unfortunately, this approach introduces a new problem: processor p may have an incorrect guess about the membership of C. We solve this by having p send a (type 1) message containing its poll-list (Poll_p) to List_p, a set of $c \log n\sqrt{n}$ randomly sampled processors. Processor p hopes that at least one processor in the set List_p will have a correct guess about C and will thus be able to forward a (type 2) message containing Poll_p to C. To prevent these processors $q \in \mathrm{List}_p$ from being flooded, each such processor q only forwards a type 2 message from a processor p if p appears in the set $\mathrm{Forward}_q$, which is a set of $\sqrt{n}$ processors that are randomly sampled in advance. Upon receiving a $< \mathrm{Poll}_p, p >$ (type 2) message from any processor q, a processor in C then

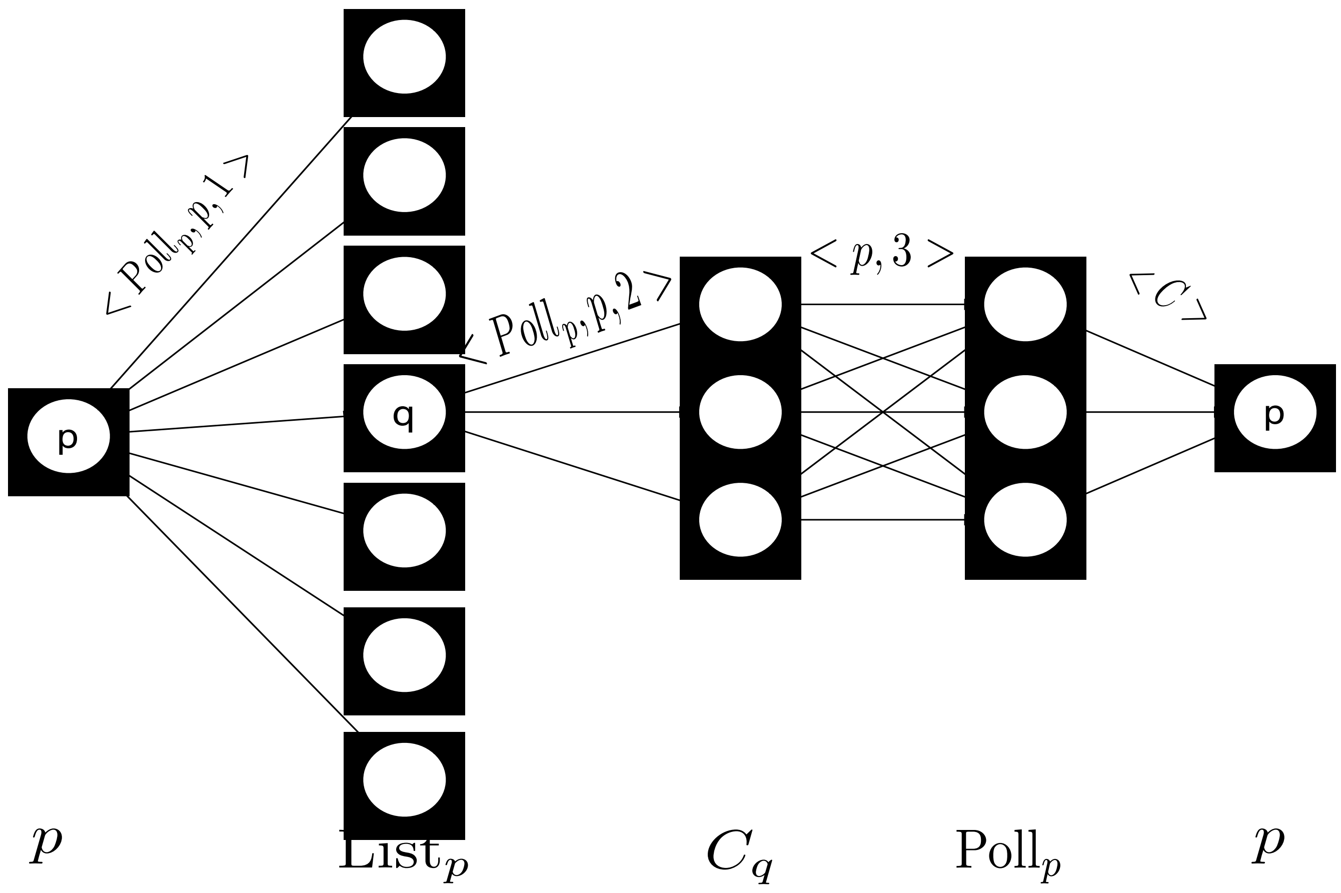


Figure 3.2: Steps 6-10 of almost-everywhere to everywhere Protocol

sends a (type 3) request with p's ID to each member $s \in \text{Poll}_p$. More precisely, a processor in C only processes the first $\sqrt{n}$ such type 2 messages that it receives from any given processor q: this is the crucial filtering that ensures that the total number of requests answered is not too large. Upon receiving a type 3 request, $< p, 3 >$ from a majority of C, s sends C_s to p, a (type 4) message.

There are two remaining technical problems. First, since a confused processor, p, can have a C_p equal to a mostly corrupt set C', C' can overload every confused processor. Hence we require that any processor, p, who receives an overload (more than $\sqrt{n}\log^2 n$) of type 3 requests wait until

```
Each processor executes the following steps in any order:
  1. Each processor $p$ selects uniformly at random, independently, and with replacement three
     subsets, $\text{List}_p$, $\text{Forward}_p$, and $\text{Poll}_p$ of processor ID's where: $|\text{List}_p| = c\sqrt{n}\log n$;
     $|\text{Forward}_p| = \sqrt{n}$; $|\text{Poll}_p| = c\log n$;
Verifying Membership in C:
  2. $member_p \leftarrow FALSE$
  3. If $p \in C_p$, then $p$ sends a message $<$ Am I in C? $>$ to the members of $\text{Poll}_p$;
  4. If $q$ receives a message $<$ Am I in C? $>$ from a processor $p \in C_q$, $q$ sends $< Yes >$ back to
     the $p$;
  5. If $p$ receives a message $<$ Yes $>$ from a majority of members of $\text{Poll}_p$ then $p$ sets $member_p \leftarrow$
     $TRUE$;
Determing C:
  6. $p$ sends a message $< \text{Poll}_p, p, 1 >$ (type 1 message) to each processor in $\text{List}_p$;
  7. For each $q$: if $< \text{Poll}_p, p, 1 >$ is the first type 1 message received from processor $p$ and
     $p \in \text{Forward}_q$, then $q$ sends $< \text{Poll}_p, p, 2 >$ (a type 2 message) to every processor in $C_q$;
  8. For each $r$: if $member_r = TRUE$ then for every processor $q$, for the first $\sqrt{n}$ type 2
     messages of the form $< \text{Poll}_p, p, 2 >$ which are received from $q$, send $< p, 3 >$ (type 3
     message) to every processor in $\text{Poll}_p$;
  9. For each $s$: for the first $\sqrt{n}\log^2 n$ different type 3 messages of the form
     $< p, 3 >$ which are each sent by a majority of processors in $C_s$, send $< C_s, 4 >$ (type 4
     message) to $p$;
 10. If $s$ receives the same type 4 message $< C', 4 >$ from a majority of processors in $\text{Poll}_s$ then
       (a) $s$ sets $C_s \leftarrow C'$; and
       (b) $s$ answers any remaining type 3 requests that have come from a majority of the current
           $C_s$, i.e. for each such request $< p, 3 >$ $s$ sends $< C_s, 4 >$ to $p$;
```

Algorithm 3.2.1: Almost-Everywhere to Everywhere

their own C_p is verified before responding. Second, the processors in C handle many more requests than the other processors. The adversary can conceivably exploit this by bombarding confused processors which think they are in C with type 2 requests. Thus, the algorithm begins with a verification of membership in C. Each processor p sends a request message to a randomly selected sample (Poll_p) which is responded to by a polled processor q if and only if $p \in C_q$. **Example:**

An example run of our algorithm is shown in Figure 3.2. This figure follows the technically challenging part of our protocol, steps 6-10, which are described in detail in Algorithm 3.2.1 listed below. In F, time increases in the horizontal direction. This figure concerns a fixed processor p that concludes $p \notin C$ in the earlier parts of the algorithm (steps 2-5). For clarity, in this example, only messages that are sent on behalf of p that eventually help p to determine C are shown. Moreover, again for clarity, we show a best case scenario where all nodes in Poll_p are assumed to have received no more than $\sqrt{n}\log^2 n$ type 3 requests. In the first step of this example, p sends the message $< \mathrm{Poll}_p, p, 1 >$ to all nodes in List_p. The node q is the only node in this set such that $p \in \mathrm{Forward}_q$, so q forwards a type 2 message of the form $< \mathrm{Poll}_p, p, 2 >$ to all the nodes in C_q. In this example, $C_q = C$. Next all nodes in C_q send the message $< p, 3 >$ to all nodes in Poll_p. In this example, all nodes in Poll_p know the set C, so they all send the message $< C >$ to p in the final step.

3.3 Detailed description of our algorithm and the election graph.

3.3.1 Committees

In a network with n processors, a committee is a collection of $O(\ln n)$ processors in the network. First, we choose a set of committees each of size $O(\ln n)$ chosen uniformly at random from the n processors in the network. We call this initial set of processors in committees layer ‘0’ committees.

Definition 3.3.1. *A committee is called* good *if less than a $1/8$ fraction of the processors in the committee have been taken over by the adversary.*

3.3.2 Committee Selection

We use the sampler to spread out coalitions of bad processors. Each committee is selected from the processors elected by the ElectHigherComm algorithm from the previous layer so that the fraction of bad committees is bounded by $\epsilon'/2\ln n$ for some $\epsilon' \in [0,1]$. The RandomID protocol, which guarantees that the additional fraction of bad processors due to elections from good committees is also bounded by $\epsilon'/2\ln n$ which then gives a total bound of $\epsilon'/\ln n$ on the growth of the fraction of bad processors appearing in successive layers in the worst case.

3.3.3 Election description

The election graph consists of a full tree with the layer '0' committees at the leaf nodes. Initially, there are committees only at the leaf nodes. These committees are created by a uniform sampler that assigns the processors to committees. The processors in non-leaf nodes will be elected by the algorithm described later in this section. This algorithm repeatedly makes use of the RandomID protocol from [6]. This protocol is used to assign random numbers in the range $[0,1]$. We use these random numbers to select a processor in each committee to advance to subsequent layers. First, we describe the properties of the protocol in [6] by the following theorem from [6].

Theorem 3.3.1. *Suppose that $|P|$ = m and there are $t < 1/6m$ adversarial peers in P. Then the* RandomID *protocol generates random keys $y_1, y_2, \ldots y_k \in \{0,1\}^s$ with $m - 2t \leq k \leq m$ and the property that for all subsets $S \subseteq \{0,1\}^s$ with $\sigma = |S|/2^s$, $E[|i\,|y_i \in S|] \in [(m-2t)\sigma, m \cdot \sigma]$ Further, the worst-case message complexity of the protocol is $O(m^2)$.*

Proof. Proof of this theorem can be found in [6]. We also note here that the Random numbers generated by the protocol in our algorithm cannot be biased. Since we assume private channels and a non-adaptive adversary, while the algorithm assumes public channels and an adaptive adversary. As the bias assumed in the generated random numbers described in the description of the algorithm comes from the adaptive behavior of adversarial nodes. □

3.3.3.1 Committee Election Protocol

The method used to run elections is a simple adaptation from the random number generation protocol of [6]. This protocol requires a bit commitment scheme h, where $h(x)$ reveals nothing about x. In practice, a cryptographic hash function should be sufficient for h.

Suppose that we have a set P of m players, $p_1, \dots p_m$, that know each other and their indexing, with any t of them being adversarial for some $t < m/6$. The round-robin random number generator works as follows for some player $p^\star \in P$ initiating it.

1. Each player $p_i \in P$ sets $P_i := P\backslash\{p_i\}$ and waits for $8i$ time steps. Each time it receives an accusation $(p_k)_{p_j}$ from a player $p_j \in P$ it has not received an accusation from yet, it sets $P_i := P\backslash\{p_k\}$. Once the $8i$ steps are over, p_i initiates the next step. p_i terminates after $8(m+1)$ steps.
2. If $|P_i| \geq 2m/3$, then p_i chooses a random $x_i \in \{0,1\}^s$ and sends $(h(x_i), P_i)_{p_i}$ to all players in P_i. Otherwise, p_i aborts the protocol (which will not happen if $t < m/6$).
3. Each player $p_j \in P_i$ receiving a message $(h(x_i), P_i)_{p_i}$ for the first time from p_i with $P_i \geq 2m/3$ chooses a random $x_j \in \{0,1\}^s$ and sends the message $(p_i, h(x_j), P_i)_{p_j}$ to p_i. Otherwise, it does nothing.
4. If all players in P_i reply within 2 time steps, then p_i sends $(\{(p_i, h(x_j), P_i)_{p_j} | p_j \in P_i\})_{p_i}$to all players in P_i. Otherwise, p_i sends an accusation $(p_j)_{p_i}$ for any $p_j \in P_i$ that did not reply correctly or in time to all players in P and stops its attempt of generating a random number.
5. Once $p_j \in P_i$ receives $(\{(p_i, h(x_k), P_i)_{p_k} | p_k \in P_i\})_{p_i}$ from p_i, p_j sends $(x_j)p_j$ to p_i.
6. If p_i gets a correct reply back from all players in P_i within 2 time steps, then it sends $(x_i, \{(x_j)_{p_j} | p_j \in P_i\})_{p_i}$ to all players in P_i and computes $y_i = x_i \oplus \bigoplus_{p_j \in P_i} x_j$ where $\bigoplus$ is the bit-wise XOR operation. Otherwise, p_i sends an accusation $(p_j)_{p_i}$ to all players in P for any $p_j \in P_i$ that did not reply correctly or in time and stops.
7. Once $p_j \in P_i$ receives $(x_i, \{(x_k)_{p_k} | p_k \in P_i\})_{p_i}, p_j$ verifies that all keys are correct. Then p_j computes $y_j^{(i)} = x_i \oplus \bigoplus_{q_k} \in P_i x_k$ and sends the message $(y_j^{(i)})_{p_j}$ to p_i.

8. If p_i receives y_i from at least $2m/3$ players in P within 2 time steps, it accepts the computation and otherwise sends an accusation $(p_j)_{p_i}$ to all players in P for any $p_j \in P_i$ that did not reply correctly or in time.

3.3.3.1.1 ALGORITHMIC DESCRIPTION. The RANDOMID protocol elects only one processor from each committee to advance to the next layer i of the network and broadcasts a message to its members informing them which processor was elected. This is done using a procedure called ELECTHIGHERCOMM(A) shown in Algorithm 3.3.3. The procedure is called once for each layer $i-1$ to elect the nodes in layer i. As each processor is elected to layer i, the processors in its respective committees in layer $i-1$ are informed via a broadcast to the processors in their committees of the election of these processors. Within the procedure ELECTHIGHERCOMM, the selected processors are assigned committees by the sampler to disperse possible coalitions of bad processors to reduce the probability of having bad processors take over a committee. The sampler selects the processors in each committee such that there are $O(\ln n)$ sized committees with only a small fraction of these committees being bad at each layer i.

The processors in each committee A ,learn which processors belong to A by contacting a set of processors called the monitoring set of A which we will refer to as $m(A)$ are an idea from [44]. The monitoring set of a committee A, is a set of layer '0' committees which know the processors assigned to A, whose identity is fixed in advance and is known to all processors. The processors in committee A, need only randomly sample $O(\ln n)$ processors in $m(A)$ to learn the processors in A with high probability. We will describe later in this section the mechanism by which these monitoring sets learn the processors in each committee.

These elections continue until layer l consisting of at least $O(\ln^3 n)$ processors, this algorithm is performed by a procedure called TREESELECT given in Algorithm 3.3.4. This procedure is later called by the BA algorithm shown in Algorithm 3.3.5. After receiving a set of $O(\ln^3 n)$ processors from TREESELECT, it runs the Byzantine agreement algorithm in [44] which we will refer to from now on as the HEAVYWEIGHT-BA on these set of processors. The agreed upon bit by the good processors is then broadcast to all other processors in the network. This is done by traversing the Election graph downward in a breadth first manner and broadcasting the identity of the processor selected to be the leader to each processor in each committee. We note here that this algorithm can be adapted easily to solve the Leader Election problem by using the RANDOMID protocol to select a leader and then broadcasting the identity of this leader downward through the Election graph as mentioned earlier. The monitoring sets are described in [44].

3.3.4 ALMOST-EVERYWHERE BYZANTINE AGREEMENT PROTOCOL

The almost-everywhere Byzantine agreement Protocol, called AEBA, works by calling the HEAVYWEIGHT-BA algorithm with the group of processors in the last layer of the election graph as input. The HEAVYWEIGHT-BA algorithm has a message complexity $O(n^2)$ on input of size n. We show that the algorithm AEBA sends no more that $O(\ln^3 n)$ messages per processor in a fully connected communication network. We show this by the following theorem.

Theorem 3.3.2. *The Byzantine agreement Protocol* AEBA *sends at most $O(n \ln^3 n)$ messages in total using a fully connected communication network (complete graph)* .

3.3.5 COMMUNICATION TREE AND MONITORING SETS

When processors advance in the election graph, processors within a committee do not know the identity of the other processors within the committee. The monitoring sets provide this information to the members of each committee. It is convenient to assume that the committee A, in layer i is the root of some tree with it's children being the committees linked to it via the processors elected

```
Input: A the set of processors in a committee in layer $i$
1: while not layer $0$ do
2:     if the processors if the processors in the child nodes $C_1, C_2, \dots C_j$ for $A$ are unknown then
3:         CALL FINDPROCESSORS($C_j$) to learn the processors in each $C_j$ that are children of $A$.
4:     committee $A$ in layer $i$, sends a message to each of it's children $C_1, C_2, \dots C_j$ in layer $i - 1$
5:     CALL INFORMMONITORINGSET on each of $C_1, C_2, \dots C_j$
```

Algorithm 3.3.1: INFORMMONITORINGSET(A)

Figure 3.3: Algorithm for informing sending messages to monitoring sets.

```
Input: $A$ the set of processors in a committee in layer $i$
1: for each processor $p \in A$ do
2:     sample uniformly at random a set $S$ of $O(\ln n)$ processors in $m(A)$
3:     poll the processors in $S$ for the identities of the processors in $A$
4:     accept via majority filtering the identities of the processors in $A$ from $S$
```

Algorithm 3.3.2: FINDPROCESSORS(A)

Figure 3.4: Algorithm for learning the identity of the processors its committee.

from layer $i - 1$. We will from now on refer to this tree as the communication tree. This communication tree is rooted at the set of $O(\ln^3 n)$ processors in layer '0' of the election graph referred to in Lemma 3.3.6. The children of some node i in layer l in the communication tree is the set of $O(\ln n)$ committees in layer $l - 1$, that is committees numbered $i \cdot \gamma, i \cdot \gamma + 1, \dots, (i + 1) \cdot \gamma - 1$ from layer $l - 1$ where $\gamma = r/s$ from the sampler properties. The leaf nodes of the communication tree are the layer '0' committees in the election graph. This scheme embeds the communication tree completely in the election graph. We assume sending messages from node A to node B to simply mean every processor in node A sending messages to every processor in node B with each processor deciding by majority filtering on the messages it received. The identity of the processors in committee A, are sent to $m(A)$ after the election of each processor to a committee using the procedure called INFORMMONITORINGSET. The procedure sends the identity of the elected

```
Input: A the set of processors in layer $i$ of the election graph
Output: $D$ the set of processors, elected to layer $i+1$ in the election graph
1: for each committee $C$ in layer $i$ do
2:    for each processor $p_i$ in $C$ do
3:       FINDPROCESSORS($C$)
4:    C selects the processor p with $p \leftarrow$ RANDOMID$(C)$
5:    add p to D
   {use the sampler to spread out the bad processors in committees}
6: $D = F(D)$
7: for each committee $C_i$ in $D$ do
8:    CALL INFORMMONITORINGSET($C_i$).
9: return D
```

Algorithm 3.3.3: ELECTHIGHERCOMM(A)

Figure 3.5: Algorithm for electing the next layer of processors.

```
Input: A the set of n processors
Output: P a set of $\ln^3 n$ processors .
1: $Com_0 \leftarrow$ A
2: for i=0 to $l$ and $|Com_{i+1}| > \ln^3 n$ do
3:    for each processor in layer $i$ {elect the processors to layer $i+1$ }
4:    $Com_{i+1} \leftarrow$ ELECTHIGHERCOMM$(Com_i)$
5: return P
```

Algorithm 3.3.4: TREESELECT(A)

Figure 3.6: Algorithm for electing a set of $\ln^3 n$ processors

processors in A to $m(A)$ by recursively sending messages through the children of node A in the communication tree.. The procedure FINDPROCESSORS is called by each processor in a committee to learn the identities of the processors in that particular committee. It does this by sampling uniformly at random $O(\ln n)$ processors in $m(A)$ to learn the identity of the processors in A. This guarantees that with high probability, these nodes know the processors in A.

```
Input: A the set of n processors
Output: Most good processors agree on the value of a bit.
1: $A_0 \leftarrow$ A
2: $A_1 \leftarrow$ TREESELECT($A_0$)
3: HEAVYWEIGHT-BA($A_1$) { Run Byzantine agreement protocol on $A_1$ $O(ln^3 n)$ subset of A }
   {inform the processors in successive layers the of the value of the bit}
4: for i=$l$ to 1 do
5:     for each node(committee) in layer i
6:     Broadcast to the processors in child nodes in layer i-1 the bit agreed upon.
7:     The processors in layer i-1 use majority filtering to accept the bit agreed upon.
```

Algorithm 3.3.5: AEBA(A)

Figure 3.7: Algorithm for Byzantine agreement.

3.3.6 DETAILED PROOFS

Lemma 3.3.1. *In the election graph, if the number of processors in layer $i-1$ is no less than $\ln^3 n$ and the fraction of bad processors is no more than $1/8 - \epsilon_0$ then, the fraction of bad committees in layer i is no more than $\epsilon'/2\ln n$.*

Proof. Using the uniform $(\theta, \epsilon, \beta)$ sampler with $\theta = \epsilon_0, \beta = 1/8 - \epsilon_0, \gamma = s/r = 1/c_1, c_1 > 1, \epsilon = \epsilon'/2\ln n$ and $(\log_2 e)(d\epsilon_0^2(1/8 - \epsilon_0))/3 > 2\ln n/c_1\epsilon' \Rightarrow d > C\ln n$ for some constant C. We see that the bounds are automatically satisfied from the properties of the sampler. □

Lemma 3.3.2. *Let G be the set of processors elected from good committees at layer i with at least $\ln^3 n$ processors in layer i, and the fraction of bad processors in layer i, is no more than f_i. Then with high probability, the fraction of bad processors in G is no more than $f_i + \epsilon'/2\ln n$.*

Proof. Let X_j be the random variable assigned to jth processor elected at layer $i-1$, such that $X_j = 1$ if the jth processor is bad and 0 otherwise with $X = \sum_{j=1}^{i=l} X_j$ where l is the number of processors in layer i, then these random variables are independent. Using Chernoff bounds $Pr(|X-\mu| \geq \delta\mu) \leq 2e^{-\delta^2\mu/3}$ with $\mu = E[X] = \beta l, \delta = \epsilon'/2\ln n$ we get $Pr(|X-\mu| \geq \beta l\epsilon'/2\ln n) \leq 2e^{-\beta l\epsilon'^2/12\ln^2 n} = 1/n^c$ for some constant c, if $l \geq \ln^3 n$. ☐

Lemma 3.3.3. *With high probability, at layer i, with the number of processors in layer $i-1$ being no less than $\ln^3 n$, the fraction of bad processors in layer i is no more than $f_i = f_0 + i\epsilon'/\ln n$ for some constant ϵ'.*

Proof. The proof is by induction on the layer i.

1. Base case $i = 0$.

   The fraction of bad processors initially at layer 0 is f_0 .
2. Inductive Step.

   We assume the statement is true for layer i, so using the Inductive Hypothesis, $f_i = f_0 + i\epsilon'/\ln n$. For layer $i+1$, using Lemma 1.3 the additional fraction of bad processors elected to layer $i+1$ due to bad committees is at most $\epsilon'/2\ln n$ applying Lemma 1.4 the additional fraction of bad processors elected from good committees is at most $\epsilon'/2\ln n$. So the total fraction of bad processors at layer $i+1$ is $f_i + \epsilon'/2\ln n + \epsilon'/2\ln n = f_i + \epsilon'/\ln n$ which is $f_0 + (i+1)\epsilon'/\ln n$.

☐

Theorem 3.3.3. *The Byzantine agreement Protocol* BA *sends at most $O(n\ln^3 n)$ messages in total using a fully connected communication network (complete graph) .*

Proof. The number of committees in layer i of the election graph is n/γ^{i+1}. The number of processors in layer i is $Cn\ln n/\gamma^{i+1}$. The election graph is of height $l = \ln nC - 2\ln\ln n - \ln\gamma/\ln\gamma$ and $C\ln n$ is the size of a committee, where $C > 0$ is some constant defined by the properties of the sampler in Lemma 2.2.

- Communication costs for informing monitoring sets for processors in layer i:
  - Each node in the communication tree needs only to learn the identity of it's immediate children in layer $i-1$ once. The identity of the processors in child nodes in the lower layers of the communication tree are already known.
  - The cost of learning the identity of a committee by sampling $10\ln n$ processors is $10C^2\ln^3 n$.
  - The cost of learning the processors for all the immediate children of the nodes in layer i is $10nC^2\ln^3 n/\gamma^i$.
  - The cost of learning all the processors in the communication tree of height l is then $10C\ln^3 n\left(nC-(\gamma\ln n)^2\right)/\gamma\left(\gamma-1\right)$.
  - The cost of sending messages down the tree for all layers $1\cdots l$ is:

$$C\ln^2 n(nC\ln\left(nC/\ln^2 n\right)(\gamma-1)+\ \gamma\ln\gamma\left(nC-\ln^2 n\right))/\ln\gamma\left(\gamma-1\right)^2$$

    .
  - So the total cost of informing the monitoring sets is: $C\ln^2 n(nC\ln\left(nC/\ln^2 n\right)(\gamma-1)+$ $\gamma\ln\gamma\left(nC-\ln^2 n\right))/\ln\gamma\left(\gamma-1\right)^2$
    $+10C\ln^3 n\left(nC-(\gamma\ln n)^2\right)/\gamma\left(\gamma-1\right)$. Which is $O(n\ln^3 n)$.
- Next, we calculate the message complexity for the election process:
  - The cost of learning the processors for all committees in all the layers of the election graph is $10C\ln^3 n\left(nC-(\gamma\ln n)^2\right)/\gamma\left(\gamma-1\right)$.
  - The cost of running the RANDOMID algorithm for all layers of the algorithm is

$$10C^2\ln^2 n\left(nC-\gamma\ln^2 n\right)/\left(\gamma-1\right)$$

  - The total cost of the election process is $O(n\ln^3 n)$.
- The cost of the Byzantine agreement Algorithm run on the processors in the last layer is $O(\ln^6 n)$.
- The cost of sending the agreed bit down to the processors in layer ‘0’ from the $O(\ln^3 n)$ processors in layer l is $C\ln^2 n\left(nC-\gamma\ln^2 n\right)/\left(\gamma-1\right)$.

- The total cost of this process is $O(n \ln^3 n)$.

$\square$

The proofs of the correctness of the algorithm to inform all the confused processors can be found in King and Saia [42].

3.4 EXPERIMENTS

3.4.1 EXPERIMENTAL SETUP

From now on we will refer to our algorithm as the NATREE algorithm. We ran our simulations using the BEA 64 bit Java 6.0 virtual machine JRockit on a machine with 8G of memory. The size of the network simulated was between 1,000 to 4,000,000 processors. In our algorithm, the parameters for the sampler were $\gamma = r/s = 60, \beta = \epsilon_0 = 1/12$ making $d \geq 50 \ln n$. We used the latest draft standards for hash functions FIPS 180-3[57] and the latest draft standards for digital signatures [58] in our measure of the actual bit complexity of our algorithm. We used hash functions of size 512 bits, and 2048 bits for digital signatures.

We simulated two algorithm in the experiments: NATREE which has been described previously; and the algorithm from [16] which we will refer to as the CKS algorithm. We simulated the NATREE algorithm with parameters set so that it can tolerate a $1/8$ fraction of bad processors. Our choice of the CKS algorithm was motivated by the fact that if seems to have the smallest message complexity of Byzantine agreement algorithms described in the literature. We compared these two algorithms along three metrics: number of messages sent, number of bits sent, and latency. The CKS algorithm requires each node to send to every other node in the network, so the asymptotic number of messages sent per node is $O(n)$. This is in contrast to $\tilde{O}(\sqrt{n})$ for the same metric for the NATREE algorithm. The latency for the CKS algorithm is a constant in contrast to the latency for the NATREE algorithm which is $O(\log n)$. The CKS algorithm can tolerate a $1/3$ fraction of faulty processors. We emphasize that this is larger than the fraction of bad processors that can be tolerated by our algorithm as simulated here. However, our interest in scalable communication costs inclines us to consider tradeoffs of fault tolerance for scalability.

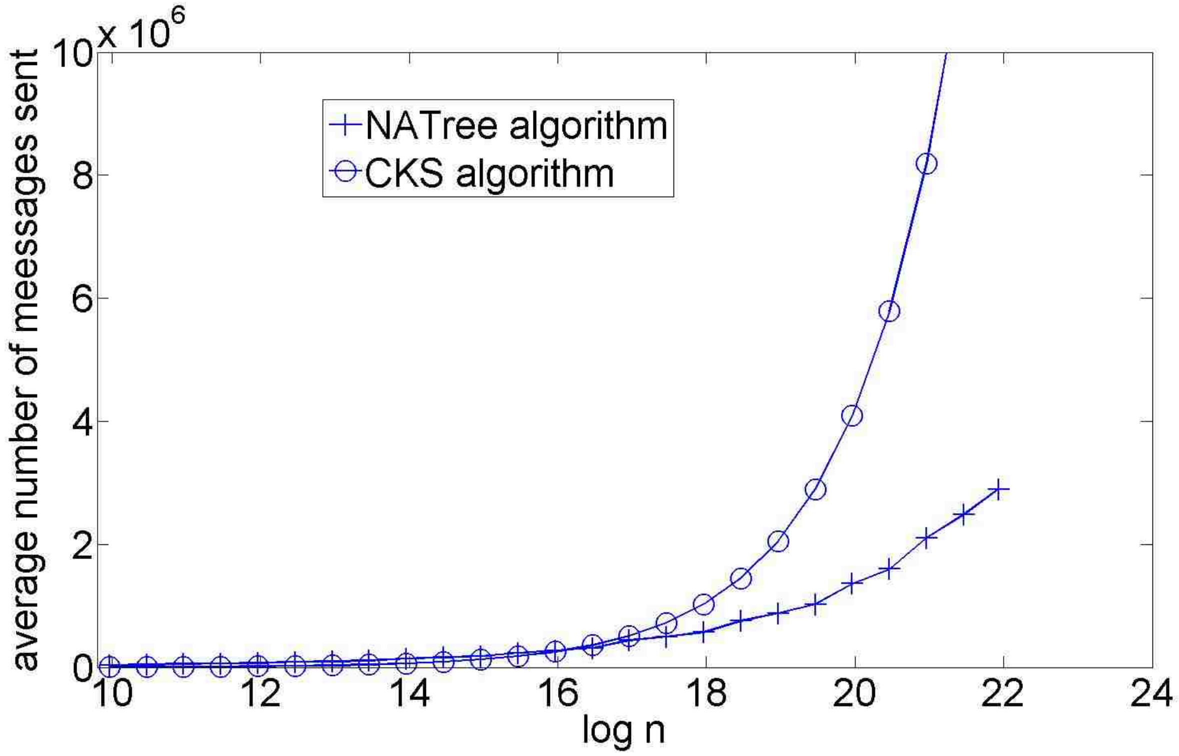
3.4.2 Experimental Results

The outcomes of our experiments are shown in Figures 3.8, 3.9 and 3.10. We note that, in our experiments, the measured message complexity for the CKS algorithm varies predictably for different network sizes. This is true since the CKS algorithm requires every node to send messages to every other node in the network a fixed number of times and then always stops. In contrast, the number of messages that a given node sends in our algorithm is less predictable. All data points shown in all of our plots are the average over at least 5 trials.

Figure 3.8 (top) shows the log of the network size vs. average number of messages sent. This plot shows that the NATREE algorithm begins to display better performance at about 65,000 processors on this metric, and for networks much larger than this size, exhibits significant improvement over the CKS algorithm. Figure 3.8 (bottom) shows the log of the network size vs. log of the average number of messages sent. Since this is a log-log plot, the slopes of the two lines fitting the data points give a good approximation to the exponents of n in the function giving the average message cost. Thus, as expected, in this plot the slope for the line for the CKS algorithm is approximately 1. Moreover, as expected, the slope for the NATREE algorithm is about $1/2$, since the almost-everywhere to everywhere part of the algorithm requires each node to send $\tilde{O}(n^{1/2})$ messages.

Figure 3.9 (top) shows the log of the network size vs the average number of bits sent. For this metric, the NATREE algorithm performs better than the CKS algorithm for all networks of size greater than about $1,000$. This is due to the larger message sizes of the CKS algorithm because of its extensive use of cryptography. The bit complexity barely registers on the graph because of the resolution and since it is at most of the order of 10^8 bits. Figure 3.8 (bottom) shows the log of the network size vs. log of the average number of messages sent. Again the CKS algorithm displays linear slope for this plot. However, the slope for our algorithm is about $1/4$, which much less than the $1/2$ expected. We believe this discrepancy is due to the fact that the "almost-everywhere" stage of the NATREE algorithm dominates in terms of the number of bits sent for network sizes we tested, and that this stage has an asymptotic cost less than $\tilde{O}(n^{1/2})$. The dominance of the almost-everywhere stage is likely due to the fact that it is the only part of the NATREE algorithm that uses cryptography. To verify our conjecture, we separated out the bit cost for the almost-everywhere stage in the plot shown in Figure 3.10 (top). As can be seen in this figure, for larger values of n the dominance of the almost-everywhere stage becomes less pronounced, and so we expect that for very large values of n, the slope in the log-log plot will approach $1/2$.

Figure 3.10 (bottom) shows the log of the network size vs latency. The latency of the CKS algorithm ($O(1)$) is better than that of the NATREE algorithm($O(\log n)$) as show by the figure. The latency for our algorithm is a step function since many values of n map to the same election graphs, and the latency of the NATREE algorithm is dominated by the diameter of the election graph.



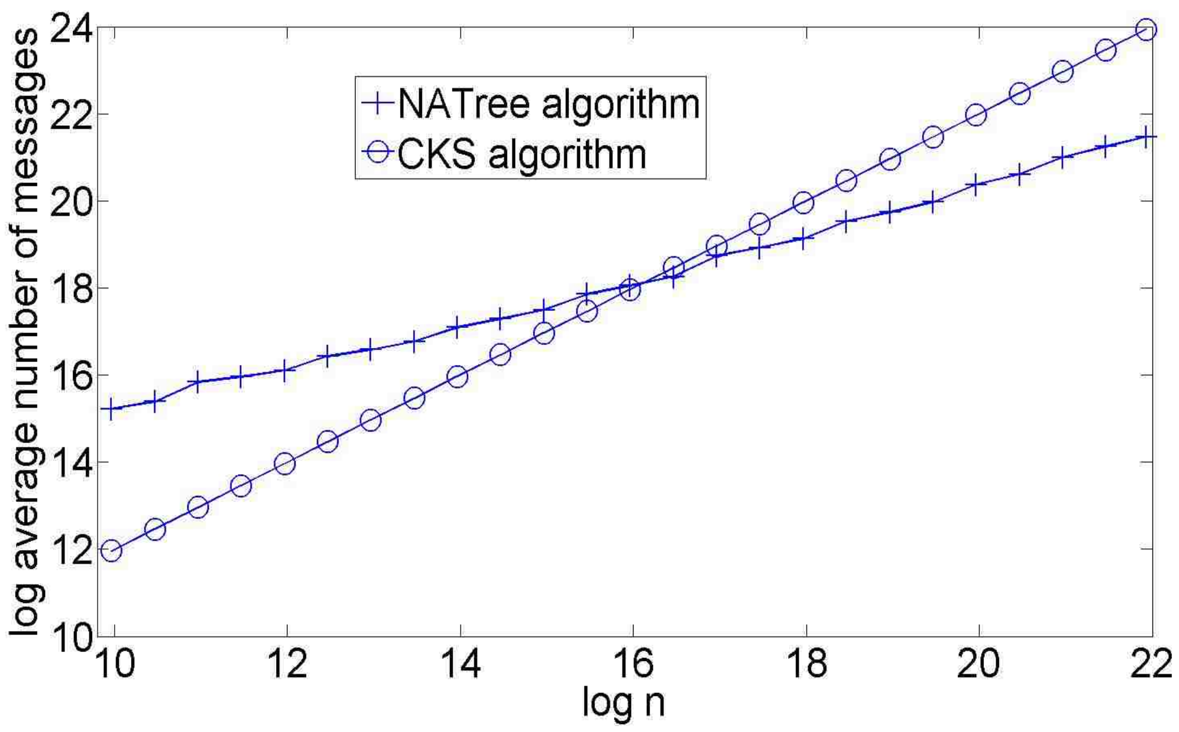
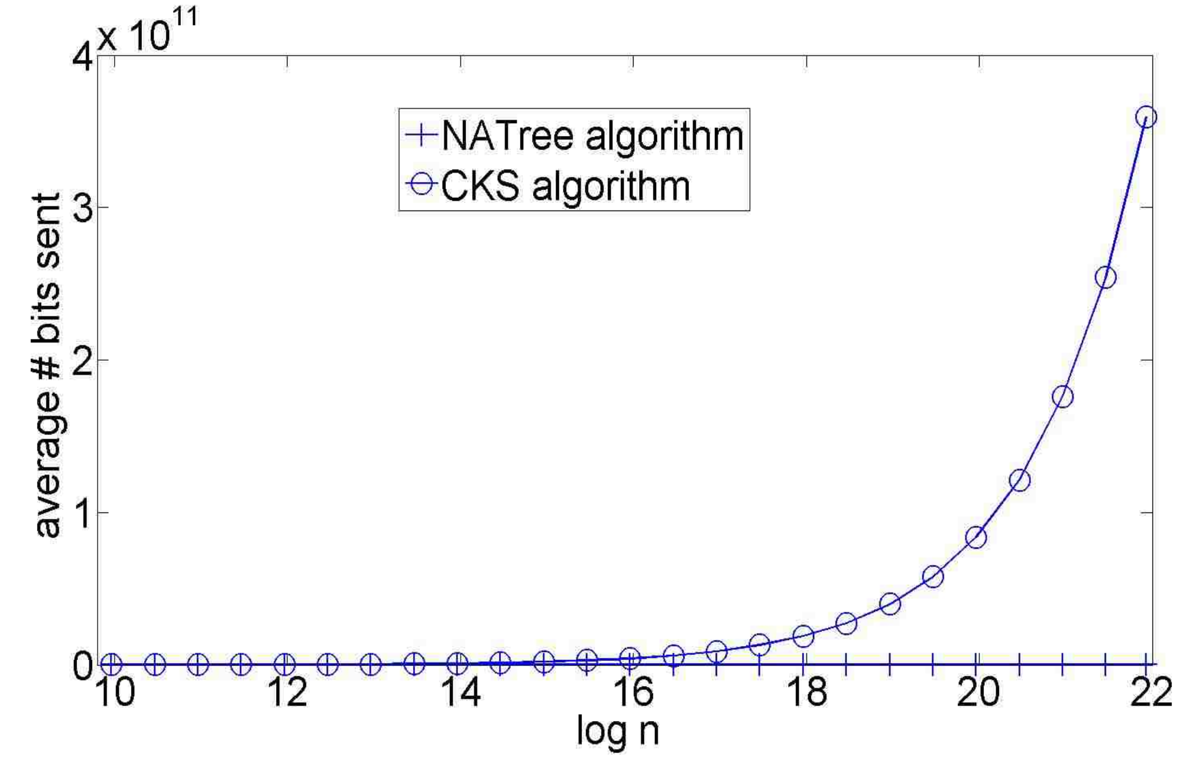


Figure 3.8: Top: Log of number of nodes vs. average number of messages; Bottom: Log of number of nodes vs log of average number of messages



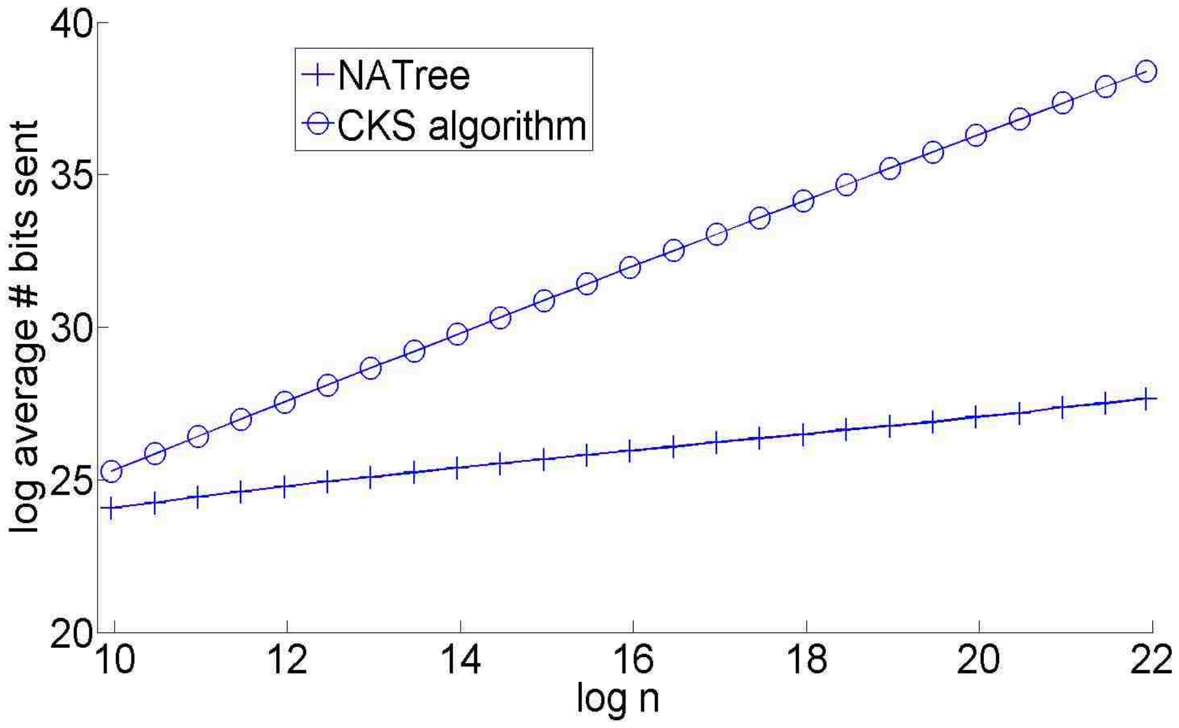
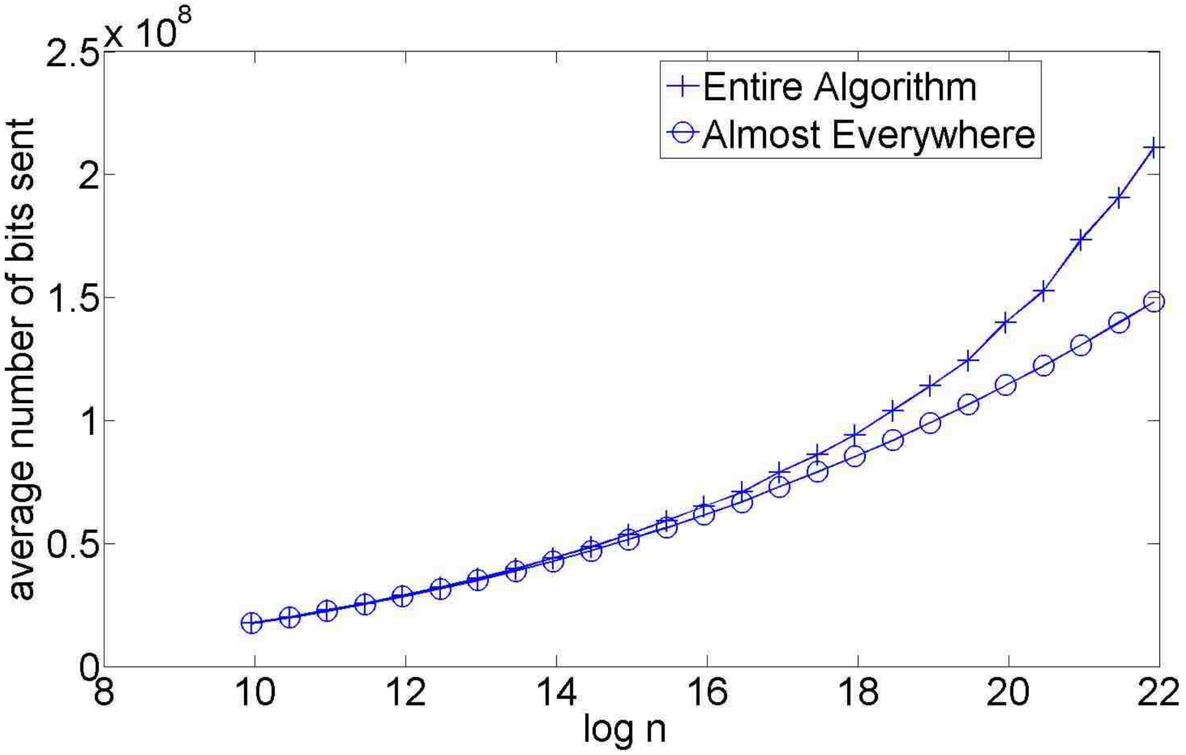


Figure 3.9: Top: Log of number of nodes vs. average number of bits sent; Bottom: Log of number of nodes vs log of average number of bits sent



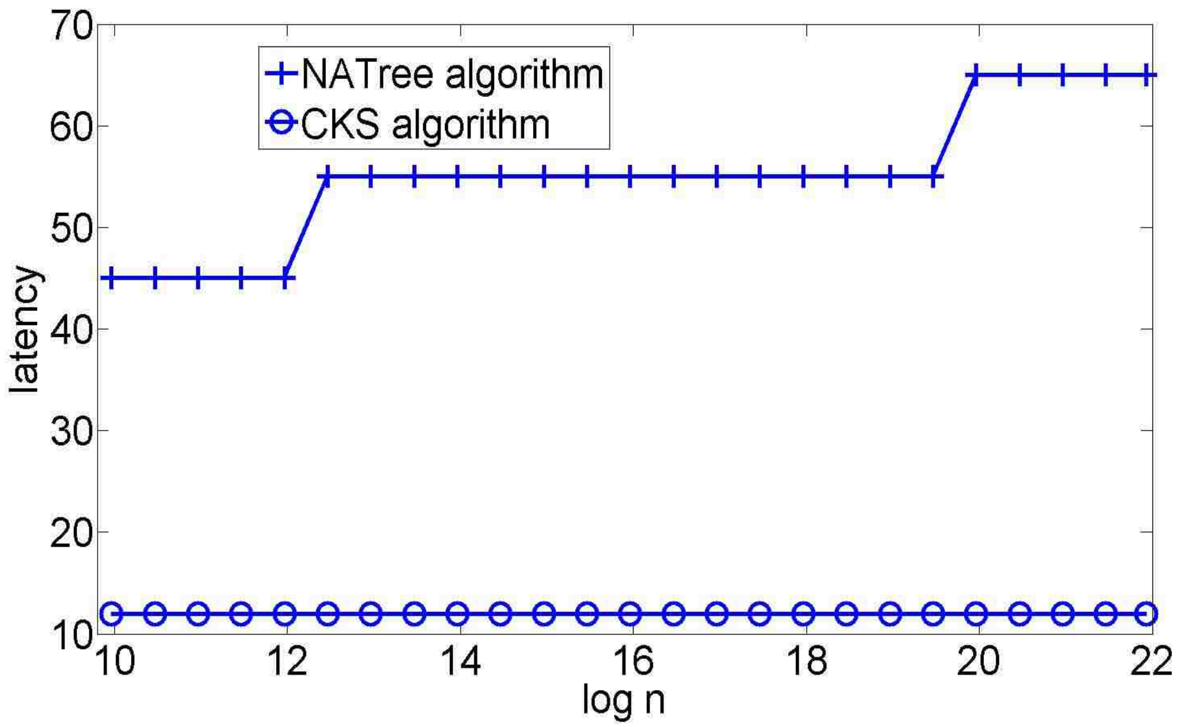


Figure 3.10: Top: Proportion of bandwidth used by the almost-everywhere part of our algorithm. Bottom: Latency vs. the logarithm of the number of nodes

Chapter 4

Adaptive Adversary

4.1 Introduction

In the previous chapter we described a Byzantine agreement algorithm for a non-adaptive adversary. Here we describe two algorithms for Byzantine agreement in the presence of an adaptive adversary. Both algorithms assume private channels, send $\tilde{O}(\sqrt{n})$ messages per processor, and have $O(\log n)$ latency. Our algorithms are necessarily Monte Carlo and succeed with high probability $1 - O(1/n^k)$ for some $k > 0$.

4.1.1 Our Contributions

Our first algorithm, which we call SHALLOWTREE, tolerates up to $\sqrt{n}$ adaptive faults and less than $n(1/6 - \epsilon)$ randomly distributed faults. The second which we call DEEPTREE, is a fully adaptive algorithm, tolerating $< n(1/3 - \epsilon)$ adaptive faults, for any positive ϵ. This algorithm is more efficient in practice than the recent result in [43].

While our SHALLOWTREE algorithm in this chapter has resource costs that are asymptotically no better than the result of [42] and the algorithm in Chapter 3 (with respect to $\tilde{O}()$ notation), in practice our algorithms are much more efficient. Our algorithms also tolerate an adaptive adversary rather than a non-adaptive adversary as the algorithms, in Chapter 3 and [42].

4.1.2 Chapter Organization

In sections 4.2 and 4.3 we give a high-level description of the two algorithms. Section 4.4 contains the analysis and proofs of correctness of the algorithms. In section 4.5 we describe our experimental results and observations and describe how these match with assumptions.

4.2 SHALLOWTREE Algorithm

4.2.1 Adversarial Model

We assume up to $\sqrt{n}$ adaptive faults and up to $n(1/6-\epsilon)$ non-adaptive randomly distributed faults. We assume the existence of a bit-commitment scheme, but otherwise make no cryptographic assumptions. We describe the algorithm, SHALLOWTREE, which has the properties given in the following theorem.

Theorem 4.2.1 (SHALLOWTREE). *The* SHALLOWTREE *algorithm tolerates up to $\sqrt{n}$ adaptive faults and $n\,(1/6-\epsilon)$ faults distributed uniformly at random. It sends $\tilde{O}(\sqrt{n})$ messages per processor, and has latency $O(\log n)$.*

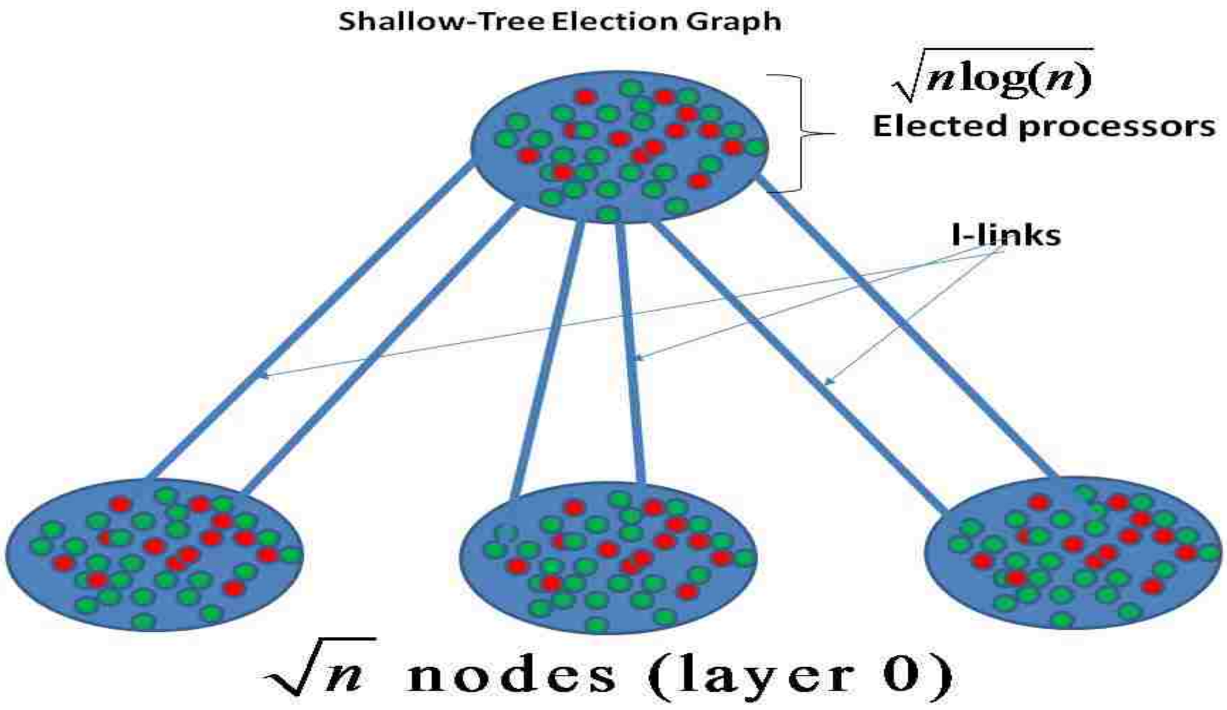


Figure 4.1: SHALLOWTREE election tree

4.2.2 Election Tree Description

The algorithm uses an election tree of height 1, and all nodes in the tree contain $\Theta(\sqrt{n}\log n)$ processors. Layer 0 contains $\Theta(\sqrt{n})$ nodes. Layer 1 contains a single node. The processors are assigned to the set of all nodes via a $(1/\log n, 1/\log n)$ sampler.

Each processor in a node in layer 1 contains links via a sampler with degree $\Theta(\log^3 n)$ to all processors in the single node at layer 0. Specifically, each node in layer 1 has a connection to its children in layer 0 via a $(1/\log n, 1/\log n)$ sampler. We call these edges *ℓ-links*. These ℓ-links perform the function of the ℓ-links in [43], they are used to send messages from the processors in the single node at layer 1 to the processors in its children at layer 0. When messages are sent from layer 1 to the processors in layer 0, the processors in layer 0 decide the message to accept by majority filtering: each processor takes the most common copy of each message.

4.2.3 High-level algorithm description

The ShallowTree algorithm is given as Algorithm 4.2.1. The algorithm makes use of the Awerbuch and Scheidler algorithm [6], which allows a set of processors to select uniformly random numbers. We will refer to this as the AS algorithm. We note that random numbers generated by the AS algorithm in the presence of an adaptive adversary are biased. The bias introduced can only "sabotage the random number generation of at most t honest players " [6]. This means only a $2t/n$ fraction of the random numbers generated are biased. This still leaves greater than a $2/3$ fraction of random numbers distributed uniformly at random since $t < n/6 - \epsilon$.

At the beginning of the algorithm, all processors in the root node run a heavy-weight Byzantine agreement algorithm (CKS [16]) to decide on an output bit. This bit is then sent down to the processors at layer 0 via the ℓ-links. At this point almost all processors have the correct bit.

The next step of the algorithm is to run the almost-everywhere to everywhere (AE2E) algorithm given as Algorithm A.1.3 in the Appendix. The global random numbers used in this algorithm will be provided by the processors at the root node. In each round, these processors will generate a random number using the AS algorithm and will send that number to the processors at layer 0 via the ℓ-links.

```
1: Processors at the root node run the CKS [16] to reach agreement on their input bits.
2: Processors at the root node sent this bit value to the processors below them via the $\ell$-links.
3: All the processors run the AE2E Algorithm A.1.3. The source of the global random numbers
   for this algorithm comes from the processors at the root node. These processors run the AS
   algorithm [6] in each round and send the result down via the $\ell$-links.
```

Algorithm 4.2.1: SHALLOWTREE

4.3 DEEPTREE Algorithm

4.3.1 Model Description

In this section, we assume an adaptive adversary where a $(1/3 - \epsilon)$ fraction of the processors may suffer Byzantine faults for any positive ϵ. We assume private channels, but otherwise make no cryptographic assumptions. We describe an algorithm DEEPTREE with the following theoretical properties. We note that DEEPTREE has equivalent theoretical properties to the algorithm in [43]. However, in practice, it is more efficient.

Theorem 4.3.1 (DEEPTREE). *The* DEEPTREE *algorithm achieves Byzantine agreement with probability* $1 - 1/n^c$ *for* $c > 0$*, against an adaptive adversary taking over up to a* $1/3 - \epsilon$ *fraction of the processors. Moreover, it has* $O(\log n)$ *latency and requires each processor to send* $\tilde{O}(\sqrt{n})$ *bits.*

4.3.2 DEEPTREE description

DEEPTREE is very similar to the algorithm of [43]. However, there are the following differences:

1. We use only 2 bins in the subcommittee election protocol. The algorithm in [43] uses $\log^{\delta} n$ bins for some $\delta > 0$.

2. The election graph is a binary tree, while the algorithm in [43] uses a q-ary tree where q is polylogarithmic.

3. We use a sampler after every election to choose the subset to elect based on the processors in the lightest bin selected by Feige's algorithm.

4. We have fewer processors in each node. Specifically, each node is of size $\Theta(\log^k n)$ where $k \geq 7/3$. In contrast to the algorithm in [43], where the size of each node is much larger.

We simply state the Lemma 3.3 from [40] which holds for the sampler used in our algorithm to choose a better committee.

Lemma 4.3.1. *Assume $r = O(\ln^k n)$ for some $k \geq 7/3$, with $s = |xr2^{-(\frac{xr}{2}-1)}|$. Then with high probability, the subcommittee produced by the $(1/\ln n, s^{-\alpha})$, $\alpha < 1$ sampler in the algorithm* SUBCOMMITTEE ELECTION *has no more than a $t/r + 1/\ln n$ fraction of bad processors.*

4.3.3 Almost-Everywhere Byzantine agreement

The algorithm for Almost-Everywhere Byzantine agreement is the same as [43], we include the algorithm in the appendix in section A.1.2. The proof of correctness is exactly the same we restate the Lemma for completeness. We call a processor knowledgeable if it is good and agrees on a message m. Otherwise, if it is good, it is confused. We assume that $(1/2 + \epsilon)n$, of the processors are knowledgeable. We use the sequential (s, t) a.e.random- source which generates a sequence of bits, with $t \geq c \log n$, and s is poly-logarithmic.

We use the protocol in Algorithm 3 of [43], we include the Lemmata describing the proof of correctness from [43] for completeness.

Lemma 4.3.2. *Almost-Everywhere Byzantine agreement algorithm (a.e.BA). In the a.e.BA algorithm, at least a $2/3 + 5l/\log n$ fraction of winning arrays are good on every level ℓ. In particular, the protocol can be used to generate a sequence of random words, of length $r = wq$ (one from each array at the root of the tournament tree) of which a $2/3 + \epsilon - 5 = /\log\log n$ fraction are random and known to $1 - 1/\log n$ fraction of good processors.*

Theorem 4.3.2. *The Almost-Everywhere Byzantine agreement algorithm sends no more than $\tilde{O}(n)$ messages.*

Proof. At each layer ℓ of the algorithm, each processors sends $\tilde{O}(n)$ messages, since we have a poly-logarithmic number of layers, the total number of messages sent is $\tilde{O}(n)$. □

Lemma 4.3.3. *Assume at the start of the protocol $n/2 + \epsilon n$ good processors agree on a message m and can generate a random bit. Let c be any positive constant. Then after a single execution of the loop:*

1. *With probability $4/(\epsilon \log n) - 1/n^c$, this protocol results in agreement on m.*
2. *With probability $1 - 1/n^c$, every processor either agrees on m or is undecided.*

4.4 Analysis and Proofs

4.4.1 SHALLOWTREE

Theorem 4.4.1. *The* SHALLOWTREE *algorithm sends at most $\tilde{O}(\sqrt{n})$ messages per processor.*

Proof. Running the subcommittee election protocol each processor sends $\tilde{O}(\sqrt{n})$ messages from running the AS algorithm in each node of layer 1. Since layer 1 has $\tilde{O}(\sqrt{n})$ nodes, the cost of running the CKS [16] algorithm is no more than $\tilde{O}(\sqrt{n})$ per processor. The cost of sending the agreed bit and the random numbers to the processors in layer 0 through the ℓ-links is $\tilde{O}(\sqrt{n})$ messages per processor. □

Lemma 4.4.1. *With probability* $1 - 1/n^c$*, after the elections at layer 0, there are at least* $5/6 + \epsilon - 1/\log n$ *fraction of good processors in layer 0.*

Proof. This follows from the sampler properties. □

Lemma 4.4.2. *With probability* $1 - n^c$*, the* SHALLOWTREE *algorithm succeeds and at least a* $5/6 + \epsilon - 2/\log n$ *fraction of the good processors learn the value of the bit. Further, after running Algorithm 4.3.3, all the good processors learn the value of the bit.*

Proof. At layer 1 of the network we have $O(\sqrt{n \log n})$ processors with high probability a $5/6 + \epsilon - 1/\log n$ fraction of the processors at layer 0 know the value of the bit after running Rabin's algorithm. Running the Algorithm 4.3.3, with high probability, from Lemma 4.3.3 all the good processors learn the value of the bit. Note that we may choose not to run Algorithm 4.3.3 in the fail-stop model. □

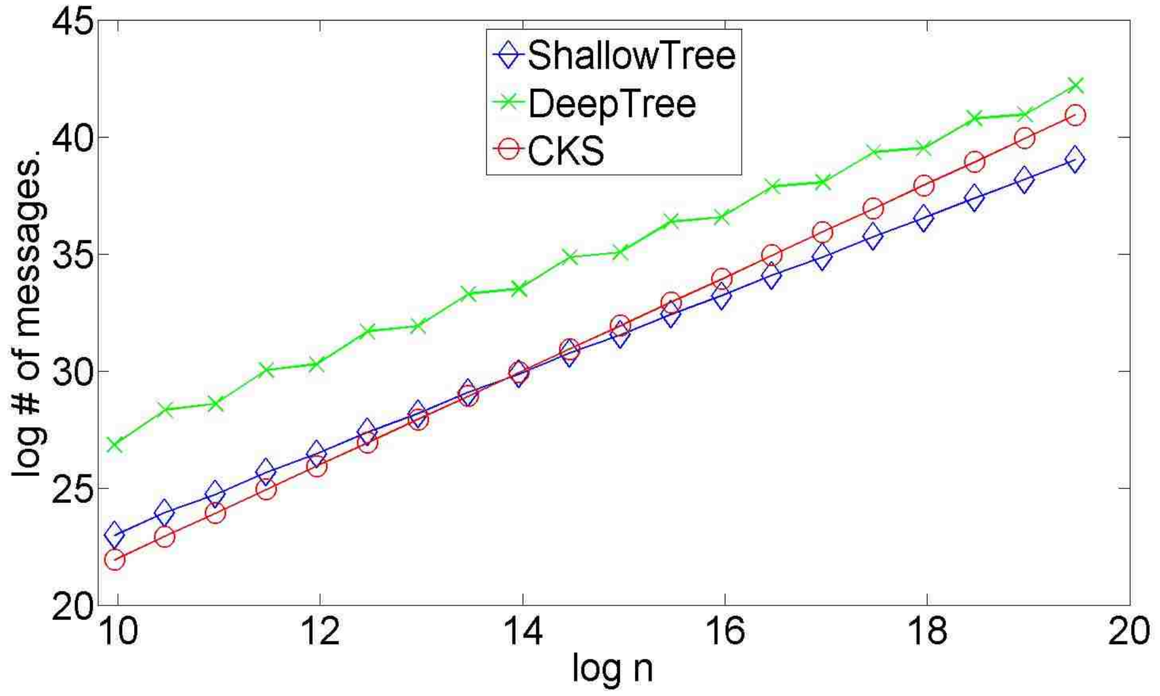
Lemma 4.4.3. *Suppose we have 2 bins in the* DEEPTREE *algorithm, let* c, ϵ, α *be any positive constants, suppose there are* r *processors with* $t = 1/3 - \epsilon$ *fraction are bad such that* $r = c_1 \log^k n$*, where* $k \geq 2 + \alpha$*, with probability* $1 - 1/n^c$*, a majority of the processors in the lightest bin are good.*

Proof. We fix some bin B. Let $X = \sum_j X_j$ be the number of good processors in bin B, with X_j the jth processor in B. These X_j are independent, with $Pr(X_j = 1) = 1/3 + \epsilon/2$,$E[X] \geq r/3 +$

$r\epsilon/2$. Applying Chernoff bounds $Pr(X < E[X] \cdot \left(1 - \frac{7}{2}\epsilon\right)) \leq \left(\frac{e^{-\frac{7}{2}\epsilon}}{\left(\left(1 - \frac{7}{2}\epsilon\right)\right)^{\left(1 - \frac{7}{2}\epsilon\right)}}\right)^{E[X]} \leq n^{-c}$ for $\epsilon = 2/\log n, \alpha \geq \frac{7}{3}, c_1 \geq 6, c \geq 1$.

In the worst case the bad processors flood the bin and with probability $1 - 1/n^c$, the good processors in bin B are no less than $(1 - 7/2\log n)\left(2 \cdot \log^{7/3} n + 6 \cdot \log^{5/3} n\right)$ which is greater than $2\log^{7/3} n - 6 \cdot \log^{5/3} n$. □

4.5 Experiments



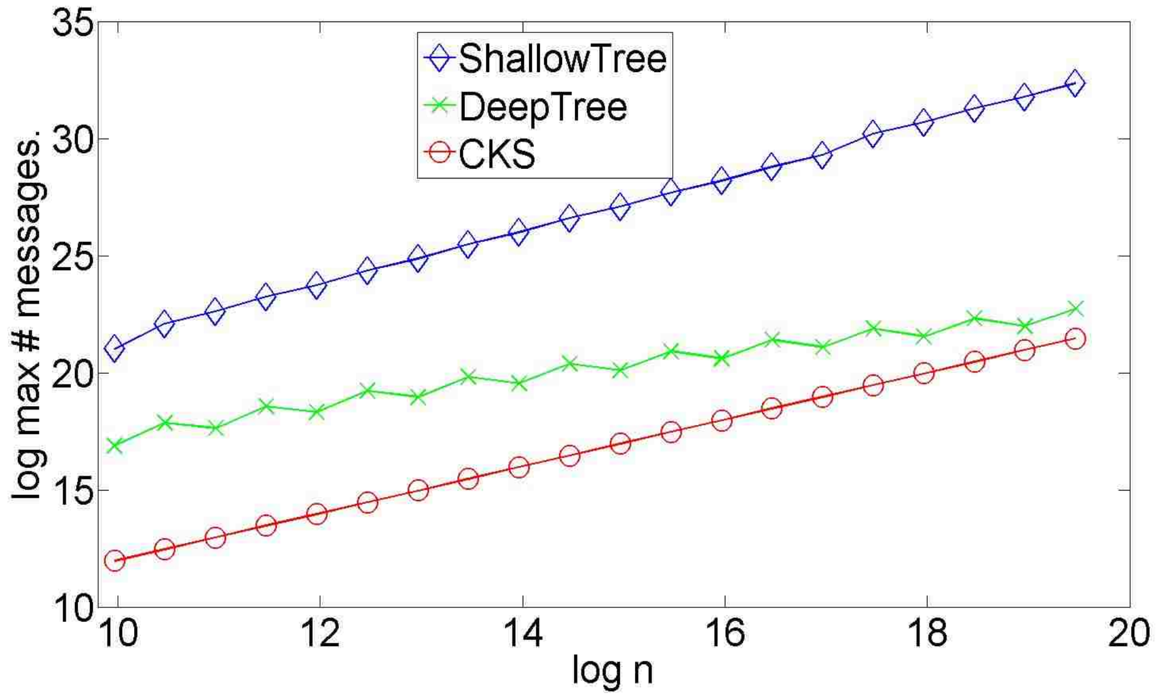
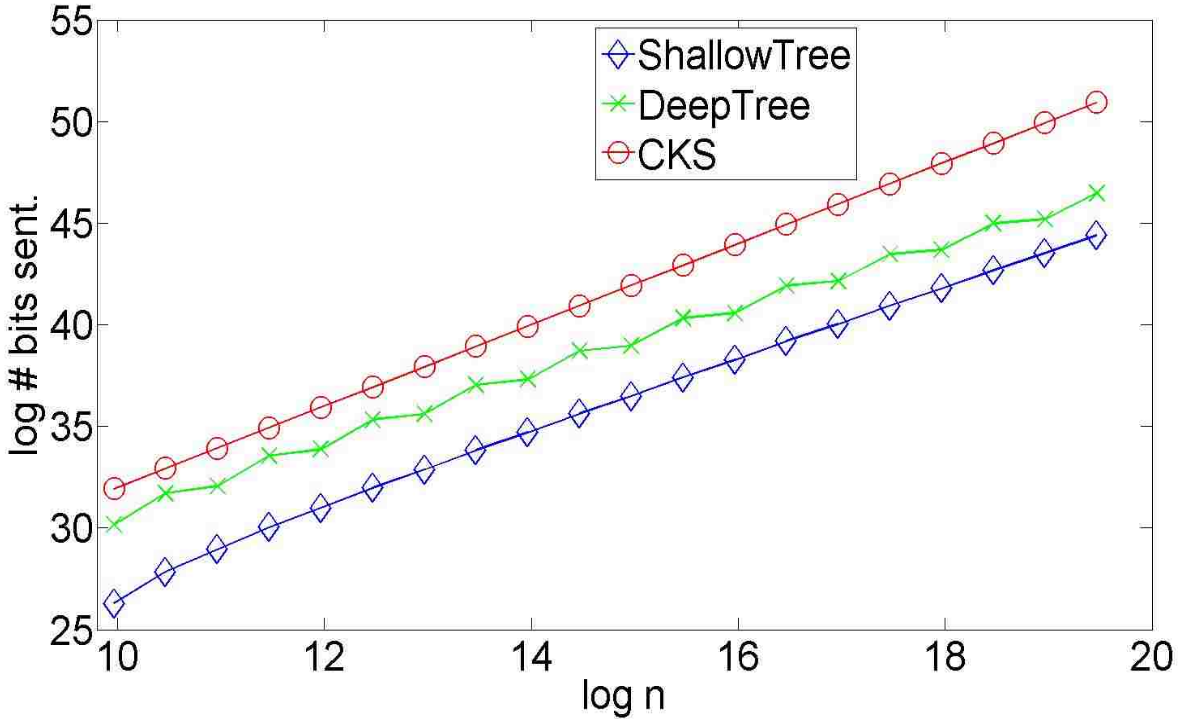


Figure 4.2: Top: Log of number of nodes vs. number of messages sent; Bottom: Log of number of nodes vs max messages sent by a node.



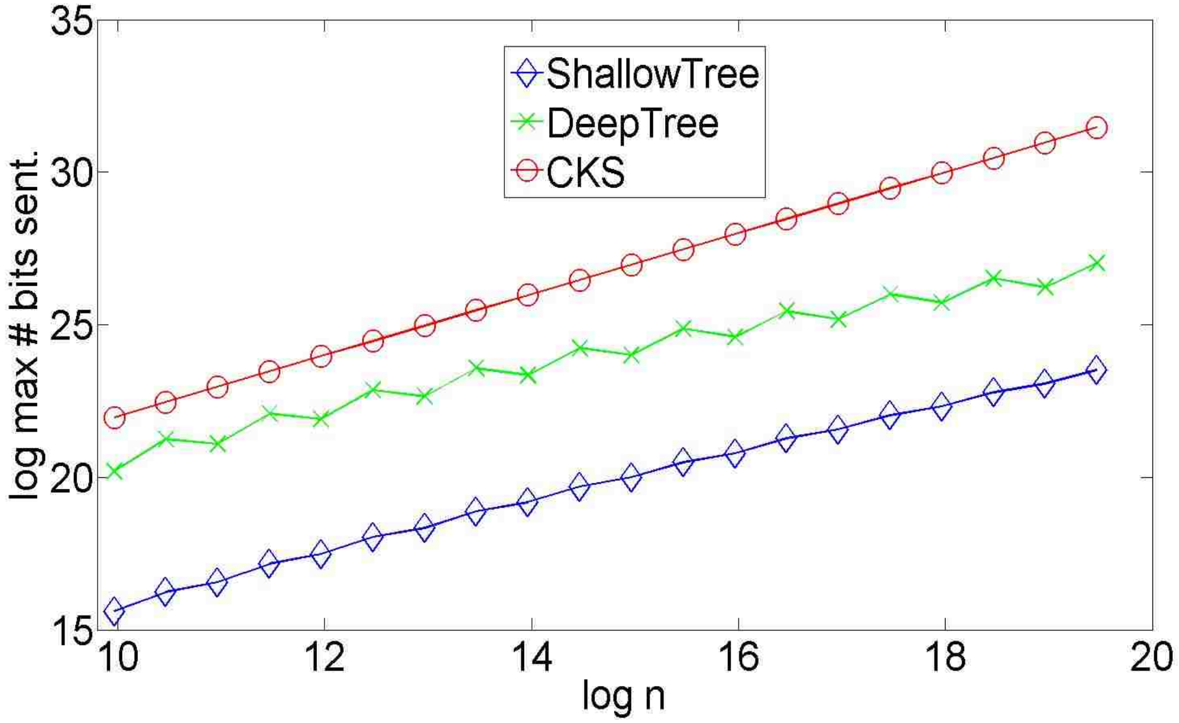


Figure 4.3: Top: Log of number of nodes vs. number of bits sent; Bottom: Log of number of nodes vs max bits sent by a node.

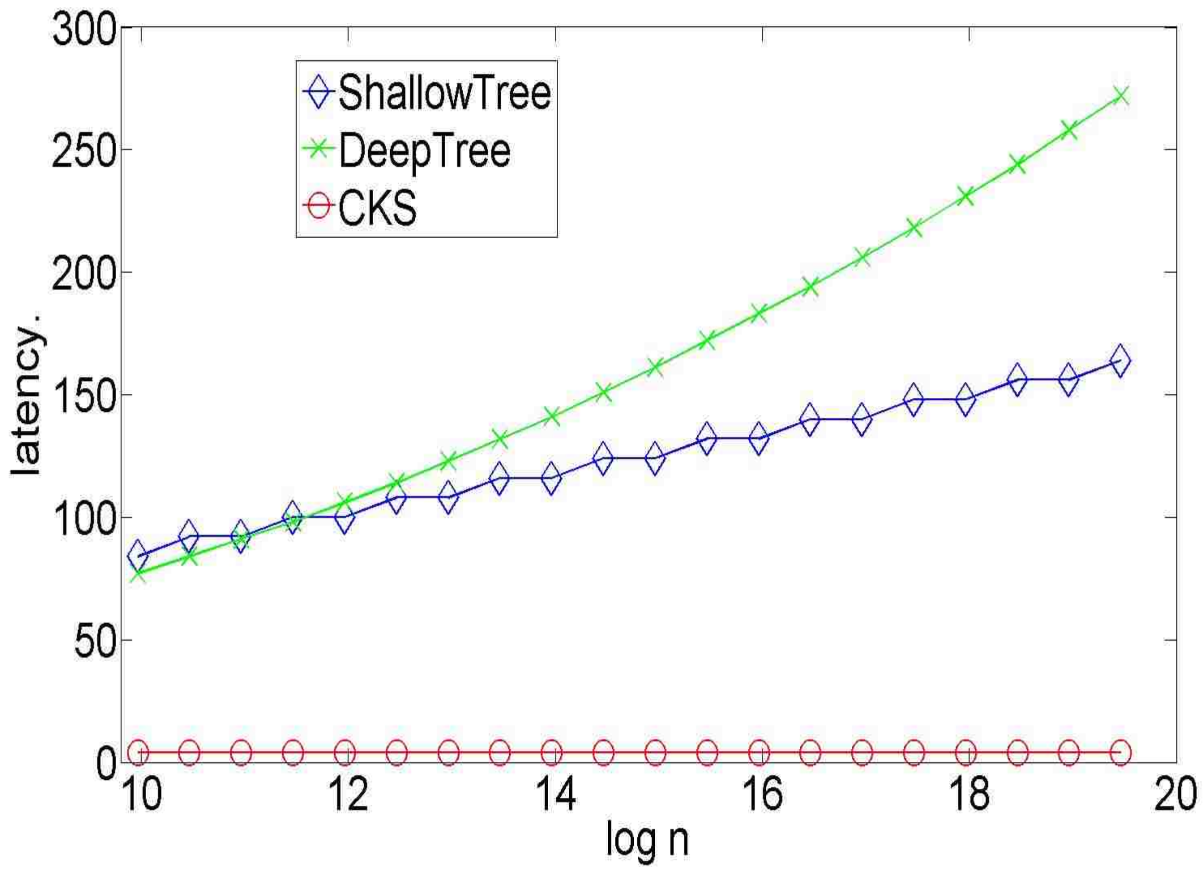


Figure 4.4: Algorithm Latency

4.5.1 Setup

Our algorithms make heavy use of samplers which we create randomly. For the DEEPTREE algorithm we could not create a complete simulation of the election tree because of memory constraints. So we estimated the cost of each step of the algorithm instead. We used the OpenMP compiler directives to speed up the simulation on a machine with 128G of memory and 48 cores. The size of the network simulated was between 1,000 and 1,024,000 processors. The latency of the DEEPTREE algorithm is $O(\log n)$, while that of the SHALLOWTREE algorithm is $O(1)$. Our goal was to measure latency, total number of bits sent, and maximum number of messages and bits sent. For the DEEPTREE algorithm because of the computational cost, we could not calculate the maximum messages/bits sent by the algorithm. As storing the data structures required is memory intensive so we use the average number of bits sent as an estimate instead, although we can calculate the total messages/bits sent.

4.5.2 Results

As seen in Figure 4.2 (top), the SHALLOWTREE algorithm sends fewer messages than the CKS algorithm, for n larger than 16,000. However, the DEEPTREE algorithm sends significantly more messages than the CKS algorithm for all network sizes tested this is likely because the constants in the algorithm are large because of the cost of running *a.e.BA-with-random-source*. Although we can extrapolate that the crossover point for the DEEPTREE algorithm beating the CKS algorithm is at about n larger than 4,000,000.

Figure 4.2 (bottom) shows that load balancing in the SHALLOWTREE algorithm is poor, with some processors sending much more than the average number of messages. This is not surprising given that the processors in the top node must send messages to a large number of processors below. In contrast, the CKS and DEEPTREE algorithms seem to have reasonably good load-balancing.

Figure 4.3 (top) shows that when we consider the number of bits sent the performance of the SHALLOWTREE and DEEPTREE algorithms is much better. The two algorithms send fewer bits than the CKS algorithm. The size of an individual message in the CKS algorithm is large since it uses digital signatures.

Figure 4.3 (bottom) shows that although the ShallowTree and DeepTree algorithms are not as well load balanced as the CKS algorithm, the maximum bits sent by an individual node is still less compared with the CKS algorithm.

Figure 4.4 shows the latencies for the ShallowTree and DeepTree algorithms are logarithmic, since we are using the same AE2E (A.1.3) algorithm in [43], although the ShallowTree algorithm performs slightly better than that of the DeepTree algorithm since the ShallowTree algorithm has only two layers in its election graph. The latency of the CKS algorithm is better than that of the ShallowTree and DeepTree algorithms since its latency is $O(1)$.

4.6 DeepTree algorithm details

4.6.1 Algorithmic Description

The processors are arranged in a binary tree with each processor appearing in a poly-logarithmic number of nodes. The nodes in higher levels contain a larger number of processors. The single node at the root contains all the processors. The leaf nodes of the network are referred to as layer 1 with its parents being the nodes in layer 2 and so on.

Each processor at the start of the algorithm possesses an array of random bits, one for each layer of the election graph network. The processors at each layer of the network use the secret sharing scheme in Section 4.6.2 to share their arrays with the nodes in layer 1. Starting from the lowest layer, the arrays in the nodes in each layer elect sub-arrays of random bits to advance to the next layer using the algorithm described in Section 4.3.2. These sub-arrays are then 'shared' using the sharing scheme among the processors in the parent node and erased from the memories of the current processors. This process is continued to the root node. The key idea is that since the parent nodes contain a larger amount of processors, it becomes harder for the adversarial nodes to learn the secret. At the root node, since we have a smaller set of bit-arrays, this will form the input to the a.e.random.source algorithm in [43] which will give the output to the protocol. The network and communication protocols are described in [43] and are included in this section for completeness.

4.6.2 Secret Sharing

We assume any secret sharing scheme which is a (n, τ) threshold scheme, for $\tau = n/3$. That is, each of n players are given shares of size proportional to the message M and τ shares are required to reconstruct M. Every message which is the size of M is consistent with any subset of τ or fewer shares, so no information as to the contents of M is gained about the secret from holding fewer than τ shares. Furthermore, we require that if a player possesses all the shares and less than $n/3$ are falsified by an adversary, the player can reconstruct the secret. See [54] for details on constructing such a scheme. We will make extensive use of the following definition.

Definition 4.6.1. SECRETSHARE*(s): To share a sequence of secret words s with n_1 processes (including itself) of which $\tau - 1$ may be corrupt, a processor (dealer) creates and distributes shares of each of the words using a (n_1, τ_1) secret sharing mechanism. Note that if a processor knows a share of a secret, it can treat that share as a secret. To share that share with n_2 processors of which at most $\tau_2 - 1$ processors are corrupt, it creates and distributes shares of the share using a (n_2, τ_2) mechanism and deletes its original share from memory. This can be iterated many times. We define a $1-$share of a secret to be a share of a secret and an $i-$share of a secret to be a share of an $(i-1)-$share of a secret.*

To reveal a secret sequence s, all processors which receive a share of s from a dealer send this shares to a processor p which computes the secret. This also may be iterated to first reconstruct $i-1$ shares from i shares, etc., and eventually the secret sequence. For completeness we include the Lemma 1 from [43]

Lemma 4.6.1. *If a secret is shared in this manner up to i iterations, then an adversary which possesses less than τ_1 $1-$shares of a secret and for $1 < j \leq i$, less than $\tau_j j-$shares of each $j-1-$share that it does not possess, learns no information about the secret.*

4.6.3 Samplers

Key to the construction of the network is the definition of an averaging sampler which was also used heavily in [45, 40] and formulated in [37, 74]. Our protocols rely on the use of averaging (or oblivious) samplers to determine the assignment of processors to nodes on each level and to determine the communication links between processors. Samplers are families of bipartite graphs which define subsets of elements such that all but a small number contain at most a fraction of ”bad” elements close to the fraction of bad elements of the entire set. Intuitively, they are used in our algorithm in order to generate samples that do not have too many bad processors. We assume either a nonuniform model in which each processor has a copy of the required samplers for a given input size, or else that each processor initializes by constructing the required samplers in exponential time. We refer the reader to the definition of samplers and technical Lemmata in section 2.5.1.

Note that limiting the degree of s as defined in 2.5.1 limits the number of subsets that any one element appears in. For this paper we will use the term sampler to refer to a $(1/\log n, 1/\log n)$ sampler, where d = $O((s/r+1)\log^c n)$ where $c \geq 7/3$.

4.6.4 Network structure

In this section we describe the network structure which is the same as that in [43] except that the sizes of the nodes and the samplers are different with $q = 2$, effectively making the network a binary tree. Let P be the set of all n processors. The network is structured as a complete binary tree. The level 1 nodes (leaves) contain $k1 = \log^c n$ processors where $c \geq 7/3$. Each node at height $l > 1$ contains $k_l = 2^l k_1$ processors; there are $(n/k^l)\log^c n$ nodes on level l and the root node at height $l^* = \log_2(n/k_1)$ contains all the processors. There are $n = 2^i$ leaves, where i is some integer, each assigned to a different processor. The contents of each node on level l is determined by a sampler where $[r]$ is the set of nodes, $[s] = P$ and $d = k_l$. The edges in the network are of three types:

1. $Uplinks$: The uplinks from processors in a child node on level l to processors in a parent node on level $l + 1$ are determined by a sampler of degree $d = 2\log^c n$, $[r]$ is the set of processors in the child node and $[s]$ is the set of processors in the parent node. Let C, C' be child nodes of a node A. Then the mapping of processors in C, C' to $[r]$ (and A to $[s]$) determines a correspondence between the uplinks of C and C'.

2. $l{-}links$: The $l{-}links$ between processors in a node C at any level $l > 1$ to C's descendants at level 1 are determined by a sampler where $[r]$ is the set of processors in the node C and $[s]$ is C's level 1 descendants. Here, $r = 2^l k_1; s = 2^l; d = O(\log^c n)$ and the maximum number of $l - links$ incident to a level 1 node is $O(k_1 \log^3 n)$.

3. Links between processors in a node are also determined by a sampler of polylogarithmic degree. These are described in a.e.BA-with-random-source.

Definition 4.6.2. *Call a node good if it contains at least a* $2 = 3 + \epsilon$ *fraction of good processors. Call it bad otherwise.*

From the properties of samplers, we have:

1. Less than a $1/\log n$ fraction of the nodes on any level are bad.

2. Less than a $1/\log n$ fraction of processors in every node whose uplinks are connected to fewer than a $2/3 + \epsilon - 1/\log n$ fraction of good processors, unless the parent or child, resp. is a bad node. We call such a set of uplinks for a processor bad.

3. If a level l node C has less than a $1/2 - \epsilon$ fraction of bad level 1 descendants, then less than a $1/\log n$ fraction of processors in C are connected through l-links to a majority of bad nodes on level 1.

4.6.5 Communication protocols

We use the following three subroutines for communication. The protocols are the same from [43]. Initially each processor p_i shares its secret with all the processors in the ith node at level 1.

SENDSECRETUP(s): To send up a sequence s of secret words, a processor in a node uses SECRETSHARE(w) to send to each of its neighbors in its parent node (those connected by uplinks) a share of each word w of s. Then the processor erases s from its own memory.

SENDDOWN(w, i): After a secret w has been passed up a path to a node C, the secret can be recovered by passing it down to the processors in the 1-nodes in the subtree. To send a secret word w down the tree, each processor in a node C on level i sends its $i-$shares of w down the uplinks it came from plus the corresponding uplinks from each of its other children. The processors on level $i - 1$ receiving the $i-$shares use these shares to reconstruct $i - 1-$shares of w. This is repeated for lower levels until all the 1-shares are reconstructed by the processors in all the 1-nodes in C's subtree. The processors in the 1-node each send each other all their shares and reconstruct the secrets received. Note that a processor may have received an $i-$share generated from more than one $i-1$ share because of the overlapping of sets (of uplinks) in the sampler.

SENDOPEN(w, l) : This procedure is used by a node C on any level l to learn a word w held by the set of level 1 nodes in C's subtree. Each processor in each level 1 node A sends w up the $l-links$ from A to a subset of processors in C. A processor in C receiving a version of w from each of the processors in a level 1 node takes a majority to determine the node A's version of w. Then it takes a majority over the values obtained from each of the level 1 nodes it is linked to.

4.6.6 Correctness of communications

We include the Definition 4 and Lemma 3 from [43].

Definition 4.6.3. *A good path up the tree is a path from leaf to root which has no nodes which become bad during the protocol.*

Lemma 4.6.2.
1. *If* SENDSECRETUP*(s) is executed up a path in the tree and if the adversary learns a word of the secret s, there must be at least one bad node on that path.*
2. *Assume that s is generated by a good processor and* SENDSECRETUP*(s) is executed up a good path in a tree to a node A on level l, followed by* SENDDOWN*(w,l) where w is a word of s, and then* SENDOPEN*(w). Further assume there are at least a $1/2+\epsilon$ fraction of nodes among A's descendants on level 1 which are good, and whose paths to A are good. Then a $1-1/\log n$ fraction of the good processors in A learn w.*

For completeness we include Algorithm 1 from [43] called A.E.BA-WITH-RANDOM-SOURCE.

4.6.6.1 A.E.BA WITH RANDOM SOURCE

Theorem 4.6.1. *[*A.E.BA-WITH-RANDOM-SOURCE*] Given a sequential (s, t)* A.E.RANDOM-SOURCE *which generates s bits in $f(s)$ time and $g(s)$ bit complexity per processor, and let C_1 and C_2 be* *any positive constants. Then there is a protocol which runs in time $O(f(s))$ with bit complexity $O(\log n + g(s))$ per processor, such that with probability at least $1 - e^{C_1 n} + 1/2^t$, all but $C_2 n/\log n$ of the good processors commit to the same vote b, where b was the input of at least one good processor.*

The algorithm is an implementation of Rabin's global coin toss Byzantine agreement protocol except that the coin toss is a.e., rather than global. In addition, only some of the coin tosses are random, and broadcast is replaced by sampling a fraction of the other processors according to the edges of certain type of $O(\log n)$ regular graph. This graph has the property that almost all of the samples contain a majority of processors whose value matches the value of the majority of the whole set.

We assume each committee is of size r with r being at least of size $O(\log^{7/3} n)$. We run Feige's algorithm to elect two candidate bins from each committee to get the candidates to advance in each layer of the Election graph.

Each processor has $r + 1$ blocks each at least of size $O(\log r)$. Let the array of blocks for each processor j be $B_j(0), B_j(1), \ldots, B_j(r)$. With the block $B_j(0)$ containing the bin choice for processor j, and some extra random bits which are described later that are used to select the actual candidates that will advance. The bits $B_j(1), \ldots, B_j(r)$ contain random bits which are used to run a.e Byzantine agreement with a random source.

Chapter 5

Byzantine agreement with a Random Beacon

5.1 Introduction

We introduce a class of algorithms based on the assumption of the existence of a Random Beacon. We show that we can use a Random Beacon to improve the efficiency of Byzantine agreement algorithms.

A Random Beacon is a random stream of bits available to all the processors. It was first defined by Rabin in [63] where it was used to build a contract signing protocol. Maurer et al. in [53, 17, 5, 30] describe a computation model called the Memory Limited or Limited Storage space Model, which uses the existence of a Random Beacon (random stream of bits) which is visible to all processors. Particularly, Aumann and Rabin in [5] describe the Limited storage space model as *"limited-storage-space model postulates an eavesdropper who can execute arbitrarily complex computations, and is only limited in the total amount of storage space (not computation space) available to him. The bound on the storage space can be arbitrarily large (e.g. terabytes), as long* *as it is fixed. Given this bound,the protocol guarantees that the probability of the eavesdropper of gaining any information on the message is exponentially small".* The existence of a Random Beacon is part and parcel of this postulate.

Lee, Clark et al. showed in [50, 23] how to build a Random Beacon using publicly available information on the Internet, specifically using information from the Dow Jones Industrial Average(DJIA) or from other financial data. Eastlake in [31, 32] also describes how to build a selection protocol out of a Random Beacon.

We further note that there may be hardware implementations of a Random Beacon. For example, a trusted node in a massively parallel computer, a wired node in a sensor network, or a satellite broadcasting random bits may be useful in a broad range of domains.

5.2 Algorithms and their Description

We now describe our algorithms that use a Random Beacon.

5.2.1 RBQUERY

The first algorithm RBQUERY is presented as Algorithm 5.2.1 (RBQUERY). This algorithm assumes $t < n(1/3 - \epsilon)$ for $\epsilon > 0$, and that all channels are private. In section 5.3.1, we prove the following theorem about this algorithm.

Theorem 5.2.1. *For any positive k, there exists sufficiently large C (in the algorithm), such that Algorithm 5.2.1 has the following properties with probability at least* $1 - 1/n^k$*:*

- *The algorithm is correct, that is each good processor terminates with the same value and this value equals the input bit of some good processor; and*
- *All good processors terminate in $O(\log n)$ rounds; and*
- *All good processors terminate in $O(1)$ rounds in expectation; and*
- *If each bad processor sends $O(\log n)$ messages per round then the total number of bits sent by all processors is $O(n \log n)$ in expectation.*

We now describe the algorithm RBQUERY. Each processor has a vote which is the bit held by that processor at the start of the round. Also in each round, a processor selects $\Theta(\ln n)$ processors uniformly at random with replacement to query for their votes. Each processor sends its vote to the processors that queried it, while it receives votes from the processors that it queried. After each round, the fraction of votes received that are for the majority bit is computed. If the fraction of processors that vote for the majority bit is $\geq (1-\epsilon_0)(2/3+\epsilon/2)$ for some ϵ_0 to be determined later, then the processor sets its vote to the majority bit. Otherwise the processor just sets its bit to the value from the Random Beacon. A processor terminates after it has 1) computed a fraction value at least $\geq (1-\epsilon_0)(2/3+\epsilon/2)$ with a majority bit value b, and 2) the global coin value equals b twice.

Note that if all processors simply commit to the Beacon, then we would not be able to solve Byzantine agreement as the bit from the Random Beacon may not be the input bit of any good processor. Secondly, it may not be the bit held by a large fraction of the good processors. These scenarios violate the conditions for Byzantine agreement.

```
Initialize:

  1. $vote \leftarrow b_i$
  2. $Match \leftarrow FALSE$

Repeat until termination:

  1. Select $C \ln n$ processors uniformly at random with replacement. Set $\mathrm{In-Neighbors}$ to these
     processors. Send request messages to all processors in $\mathrm{In-Neighbors}$
  2. Receive all request messages. Set $\mathrm{Out-Neighbors}$ to all processors from which request
     messages were received
  3. Send vote to all processors in $\mathrm{Out-Neighbors}$
  4. Collect votes from all processors in $\mathrm{In-Neighbors}$
  5. $coin \leftarrow$ next output of random beacon
  6. If $Match$ then
      (a) If $coin = vote$ then commit to value $vote$ and **terminate**
  7. Else
      (a) $maj \leftarrow$ majority bit among $\mathrm{Out-Neighbors}$
      (b) fraction $\leftarrow$ fraction of votes received for $maj$
      (c) If $fraction \geq (1 - \epsilon_0)(2/3 + \epsilon/2)$ then
          i. $vote \leftarrow maj$
          ii. If $coin = vote$ then $Match \leftarrow TRUE$
      (d) else
          i. If $coin$ = "heads", then $vote \leftarrow 1$, else $vote \leftarrow 0$;
```

Algorithm 5.2.1: RBQUERY

We will analyze the correctness of the algorithm in Section 5.3.1.

5.2.2 RBSAMPLER

We now describe the algorithm RBSAMPLER. The main difference in this algorithm, compared to RBQUERY, is that we make use of a sampler in determining the communication graph among the processors. RBSAMPLER is thus non-uniform in the sense that there is a different version of the algorithm for each value of n. Also, the neighbors of each processor are fixed and do not change between rounds unlike RBQUERY. Also, unlike the RBQUERY, RBSAMPLER only achieves almost-everywhere agreement.

The algorithm makes the following assumptions: The adversary is adaptive, bad nodes can send any number of messages per round and we have a full information communication model. The algorithm tolerates $t < n(1/3 - \epsilon)$ for $\epsilon > 0$ adaptive adversarial faults. In section 5.3.2, we prove the following theorem about RBSAMPLER.

Theorem 5.2.2. *For any positive k, there exists sufficiently large C (in the algorithm), such that Algorithm 5.2.2 (*RBSAMPLER*) has the following properties with probability at least $1 - 1/n^k$ for almost all processors i.e. all but $O(\log^{d-1} n)$*

- *The algorithm is correct: almost all processors terminate with the same value and this value equals the input bit of some good processor; and*
- *Almost all processors terminate in $O(\log n)$ rounds; and*
- *Almost all processors terminate in $O(1)$ rounds in expectation; and*
- *Almost all processors send polylog(n) bits.*

```
Initialize:

  1. $vote \leftarrow b_i$
  2. $Match \leftarrow FALSE$
  3. Set $\mathrm{In}-\mathrm{Neighbors}$ (resp. $\mathrm{Out}-\mathrm{Neighbors}$) to all processors that have in-edges (resp. out-
     edges) to this processor in the sampler.

Repeat until termination:

  1. Send vote to all processors in $\mathrm{Out}-\mathrm{Neighbors}$
  2. Collect votes from all processors in $\mathrm{In}-\mathrm{Neighbors}$
  3. $coin \leftarrow$ next output of random beacon
  4. If $Match$ then

       (a) If $coin = vote$ then commit to value $vote$ and **terminate**

  5. Else

       (a) $maj \leftarrow$ majority bit among $\mathrm{Out}-\mathrm{Neighbors}$
       (b) fraction $\leftarrow$ fraction of votes received for $maj$
       (c) $fraction \leftarrow$ fraction of votes received for $maj$
       (d) If $fraction \geq (1-\epsilon_0)(2/3+\epsilon/2)$ then

             i. $vote \leftarrow maj$
            ii. If $coin = vote$ then $Match \leftarrow TRUE$

       (e) else

             i. If $coin$ = “heads”, then $vote \leftarrow 1$, else $vote \leftarrow 0$;
```

Algorithm 5.2.2: RBSAMPLER

5.3 Analysis and proofs

5.3.1 Analysis and proofs for RBQUERY

For a fixed round, let $b' \in \{0,1\}$ be the bit that the majority of good processors vote for in that round. Let S' be the set of good processors that will vote for b' and let $f' = |S'|/n$. Let $0 \leq \epsilon_0 \leq 1$ be a fixed constant to be determined later. We call a processor *informed* for the round if the fraction value for that processor obeys the following inequalities:

$$(1-\epsilon_0)f' \leq fraction \leq (1+\epsilon_0)(f' + 1/3 - \epsilon)$$

Lemma 5.3.1. *For any positive integers C' and k, there exists a sufficiently large C in Algorithm 5.2.1 (*RBQUERY*) above, such that all good processors are informed for the first k rounds with probability at least $1 - 2k/n^{C'}$.*

Proof. Fix a round r. Let G be a bipartite multigraph induced by the $\mathrm{In}-\mathrm{Neighbors}$ and $\mathrm{Out}-\mathrm{Neighbors}$ selection process defined in the algorithm for round r. Note that G is constructed as follows. There are n nodes on the left hand side and copies of all these nodes on the right hand side. The adversary chooses a subset of $1/3 - \epsilon$ of the nodes on the left that are bad and each good node on the right chooses $C \log n$ neighbors on the left uniformly at random with replacement. Fix a round r and the set S'. We know that S' is of size at least $(1/3 + \epsilon/2)n$ since at least half of the good processors must vote for the majority bit. Let $f' = |S'|/n$.

Fix a good node, p, on the right hand side of the graph. Note that $\mathrm{In}-\mathrm{Neighbors}$ is chosen independently of the set S'. Let X be the number of edges from p into the set S'. Note that $E(X) = f'C\ln n$. Moreover, each edge from p falls into some processor in S' independently with probability f'. Thus, we can apply Chernoff bounds to say that for any positive ϵ_0,

$$Pr(X < (1-\epsilon_0)E[X]) < e^{-E[X]\epsilon_1^2/2} = e^{-(f'C\epsilon_0^2/4)\ln n}.$$

Similarly, we can use Chernoff bounds to say that

$$Pr(Y > (1+\epsilon_0)E[Y]) < e^{-E[Y]\epsilon_1^2/3} = e^{-(f'C\epsilon_1^2/6)\ln n}.$$

Hence the probability that either of these bounds is violated is no greater than the sum of these two probabilities, or less than $2/n^{C_1}$ for any constant C_1, where $C = 18C_1/{\epsilon_0}^2$. Let $\xi_{p,r}$ be the event that either of these bounds is violated for fixed processor p in some fixed round r. Further let ξ be the even that either of these bounds is violated for any good processor p in any of the first k rounds. Then by a union bound, we have that

$$\begin{aligned} Pr(\xi) &= \sum_{p,r} Pr(\xi_{p,r}) \\ &\leq (nk)2/n^{C_1} \\ &\leq (2k)/n^{C_1-1} \end{aligned}$$

Where the last equation holds provided that C is sufficiently large, but depending only on ϵ_0. □

In the following lemmas, we assume all good processors are informed in all rounds.

Lemma 5.3.2. *Assume all good processors have vote value equal to b at the beginning of some round r. Then all good processors will have vote value equal to b in all remaining rounds.*

Proof. If all good processors have vote value equal to b at the beginning of round r, it means that $f' = |S'|/n = 2/3 + \epsilon$. Since all processors are informed in round r, it means that for each processor, $fraction \geq (1-\epsilon_0)f' \geq (1-\epsilon_0)(2/3+\epsilon)$. Thus, each processor in round r will set its vote to the majority value, which equals b. The same argument holds for all remaining round in which all good processors are informed. □

Lemma 5.3.3. *For any round r, let S_f be the set of good processors in round r that have $fraction \geq (1-\epsilon_0)(2/3+\epsilon/2)$. Then, at the end of round r, all processors in S_f will have the same vote value*

Proof. We show this by contradiction. Assume there are two processors, x and y, where $fraction_x$ ($fraction_y$) are the fraction values of x (y, resp.), such that both $fraction_x$ and $fraction_y$ are greater than or equal to $(1-\epsilon_0)(2/3+\epsilon/2)$, and x sets its vote to 0 at the end of the round, while y sets its vote to 1.

Let f_0' (f_1') be the fraction of good processors that vote for 0 (1) during the round. Then we have that $fraction_x \geq (1-\epsilon_0)(2/3+\epsilon/2)$. By the definition of informed, we also know that $fraction_x \leq (1+\epsilon_0)(f_0' + 1/3 - \epsilon)$. This implies that

$$(1-\epsilon_0)(2/3+\epsilon/2) \leq (1+\epsilon_0)(f_0' + 1/3 - \epsilon).$$

Isolating f_0' in this inequality, we get that

$$f_0' \geq \frac{1/3 + (3/2)\epsilon - \epsilon_0 + (1/2)\epsilon\epsilon_0}{1+\epsilon_0}.$$

A similar analysis for $fraction_y$ implies that

$$f'_1 \geq \frac{1/3 + (3/2)\epsilon - \epsilon_0 + (1/2)\epsilon\epsilon_0}{1 + \epsilon_0}.$$

But then we have that,

$$\begin{aligned} f'_0 + f'_1 &\geq \frac{2/3 + 3\epsilon - 2\epsilon_0 + \epsilon\epsilon_0}{1 + \epsilon_0} \\ &> 2/3 + \epsilon \end{aligned}$$

where the last line is clearly a contradiction that holds provided that $\epsilon_0 < (3/4)\epsilon$. □

Lemma 5.3.4. *Assume the first round in which some good processor commits to a value is round r, and assume that some processor commits to value b in that round. Then all good processors that commit in rounds r or later will commit to the value b.*

Proof. Consider a processor p that commits to bit value b in round r. Then processor p's $Match$ value must be equal to true in round r. This means there must have been some previous round, r', in which p's $Match$ value was first set to true. Among processors that commit to values in round r, let p be a processor with the smallest such r' value. Note that at the point that $Match$ was set to true in round r', p's vote value must have been b, since a processor's vote value can not change after its $Match$ value is set to true.

Let S_f be the set of good processors in round r' that have $fraction \geq (1 - \epsilon_0)(2/3 + \epsilon/2)$. By Lemma 5.3.3, all processors in S_f set their vote value to b at the end of the round. Now since processor p set $Match$ to true in round r', it must be the case that the outcome of the global coin in that round was equal to b. This means at the end of round r', *all* good processors set their vote values to b. But then, by Lemma 5.3.2, for all rounds subsequent to round r', all processors will have vote values equal to b and so if they commit to any value, it will be the value b. □

Lemma 5.3.5. *In any round r, with probability at least $1/2$, at the end of that round, all good processors will have the same vote value.*

Proof. Let S_f be the set of good processors in round r that have $fraction \geq (1-\epsilon_0)(2/3+\epsilon/2)$. By Lemma 5.3.3, all processors in S_f will set their vote value to the same value, call it b, at the end of the round. But with probability $1/2$, the global coin in round r will have value b, and all the remaining good processors will set their vote value to b. □

Lemma 5.3.6. *Consider any round r in which all processors have the same vote value at the beginning of the round. Then the expected number of remaining rounds before all processors terminate is no more than 4.*

Proof. By Lemma 5.3.2, in all remaining rounds, all good processors will have the same value, and so all good processors will have $fraction \geq (1-\epsilon_0)(2/3+\epsilon/2)$. Thus, there are at most two events that must occur before any good processor terminates: the global coin must match the processor's vote value twice. The expected number of rounds until these two events occur is 4. □

We now prove Theorem 5.2.1 below.

Proof. By Lemmas 5.3.5, 5.3.6, 5.3.2 and 5.3.1, all processors terminate in $O(\log n)$ rounds with probability at least $1-1/n^k$ for any fixed positive k and all processors terminate in $O(1)$ rounds in expectation. Since the bad processors each send at most $O(\log n)$ messages per round, it follows that the total number of bits sent by all processors is $O(n \log n)$ in expectation. Finally, the fact that the algorithm is correct with probability at least $1-1/n^k$ follows from Lemmas 5.3.1, 5.3.2 and 5.3.4. □

5.3.2 Analysis and proofs for RBSAMPLER

For a fixed round, let $b' \in \{0, 1\}$ be the bit that the majority of good processors vote for in that round. Let S' be the set of good processors that will vote for b' and let $f' = |S'|/n$. Let $0 \leq \epsilon_0 \leq$ be a fixed constant to be determined later. We call a processor *informed* for the round if the processor is good and the fraction value for that processor obeys the following inequalities:

$$(1 - \epsilon_0)f' \leq fraction \leq (1 + \epsilon_0)(f' + 1/3 - \epsilon)$$

The proof of the following lemma is equivalent to that in King and Saia [43].

Lemma 5.3.7 (Graph existence)**.** *For any positive k and positive constants ϵ_0 and d, there exists a directed multigraph G on k vertices with maximum out-degree no more than $C_3 \log^d n$, where C_3 depends only on ϵ_0 and d, such that if this communication graph is used in Algorithm 5.2.2 (*RBSAMPLER*), then in every round all but a $1/\log^d n$ fraction of the good processors are informed.*

The following simple corollary follows by summing up the number of processors that are not informed in every round.

Corollary 5.3.1. *Let C' be an positive integer. If the conditions of Lemma 5.3.1 are met then all but a $C'/\log^{d-1} n$ fraction of the good processors are informed in every one of the first $C' \log n$ rounds.*

We will say that a processor is *always informed* if it is informed for every round from the start of Algorithm 5.2.2 (RBQUERY), to the round in which the processor terminates.

The proof of the following lemma is identical to the proof of Lemma 5.3.4 in the previous section.

Lemma 5.3.8. *For any round r, let S_f be the set of informed processors in round r that have $fraction \geq (1-\epsilon_0)(2/3+\epsilon/2)$. Then, at the end of round r, all processors in S_f will have the same vote value.*

The proof of the following lemmas are the same as in the previous section.

Lemma 5.3.9. *Assume the first round in which some always informed processor commits to a value is round r, and assume that some always informed processor commits to value b in that round. Then all always informed processors that commit in rounds r or later will commit to the value b.*

Lemma 5.3.10. *Assume all always informed processors have vote value equal to b at the beginning of some round r. Then all always informed processors will have vote value equal to b in all their remaining rounds.*

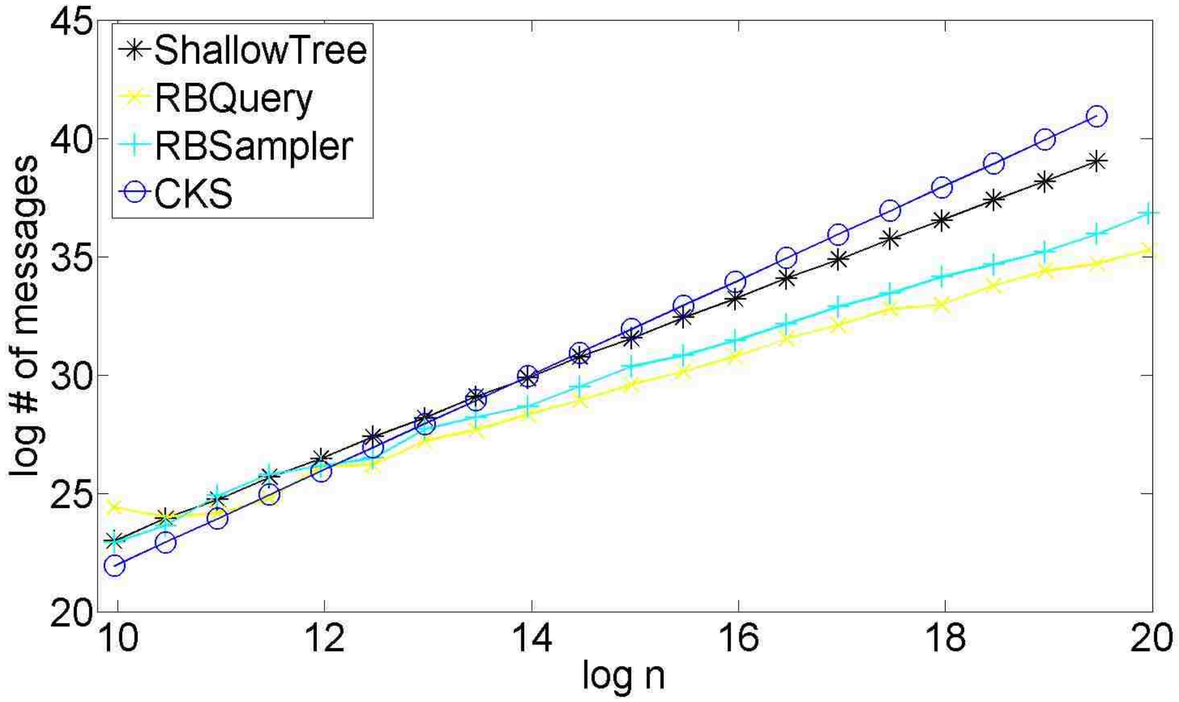
Lemma 5.3.11. *In any round r, with probability at least $1/2$, at the end of that round, all informed processors will have the same vote value.*

Lemma 5.3.12. *Consider any round r in which all always informed processors have the same vote value at the beginning of the round. Then the expected number of remaining rounds before all always informed processors terminate is no more than 4.*

We now prove Theorem 5.2.2.

Proof. By Lemmas 5.3.11, 5.3.12, 5.3.10 and 5.3.7, almost all processors terminate in $O(\log n)$ rounds with probability at least $1-1/n^k$ for any fixed positive k and all processors terminate in $O(1)$ rounds in expectation. Since the bad processors each send at most $O(\log n)$ messages per round, it follows that the total number of bits sent by all processors is $O(n \log n)$ in expectation. Finally, the fact that the algorithm is correct with probability at least $1 - 1/n^k$ follows from Lemmas 5.3.7, 5.3.10 and 5.3.9. $\square$

5.4 Experimental Results



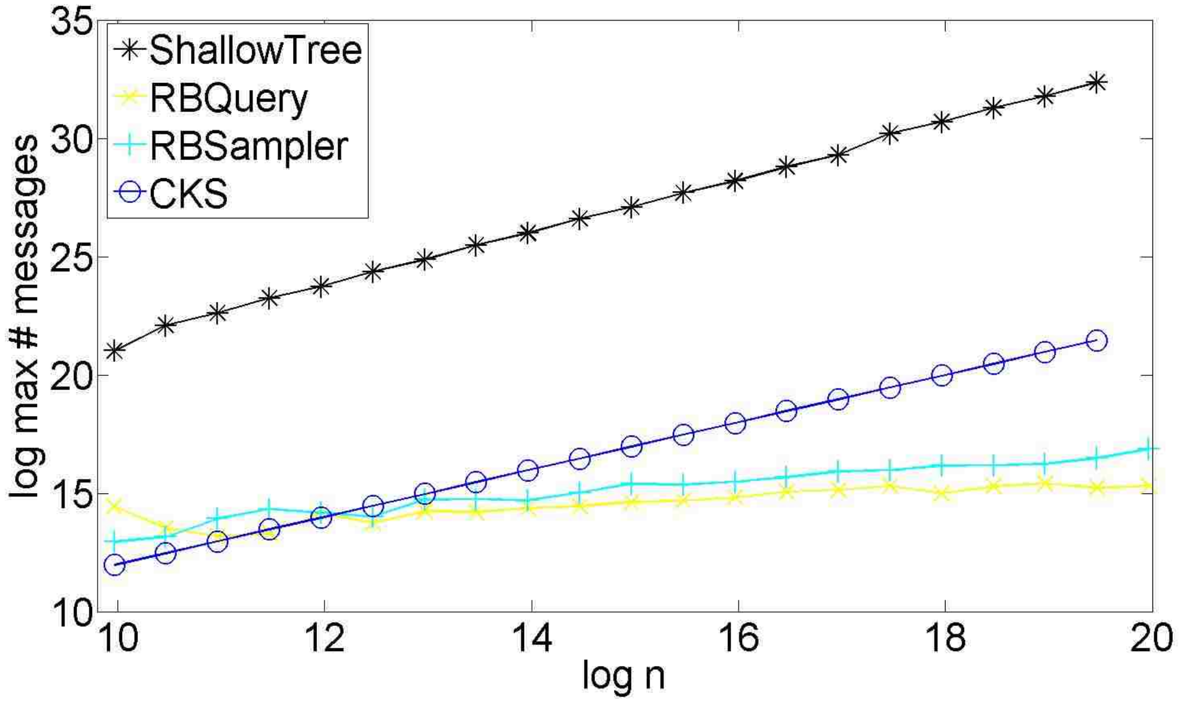
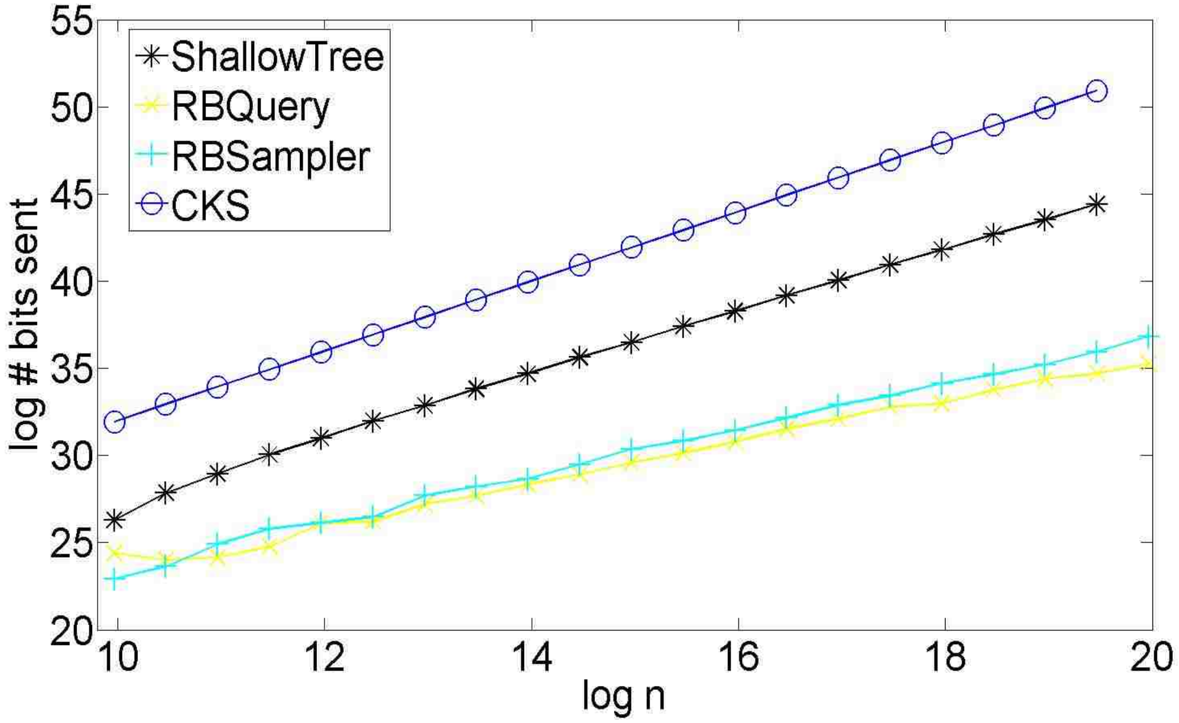


Figure 5.1: Top: Log of number of nodes vs. number of messages sent; Bottom: Log of number of nodes vs max messages sent by a node.



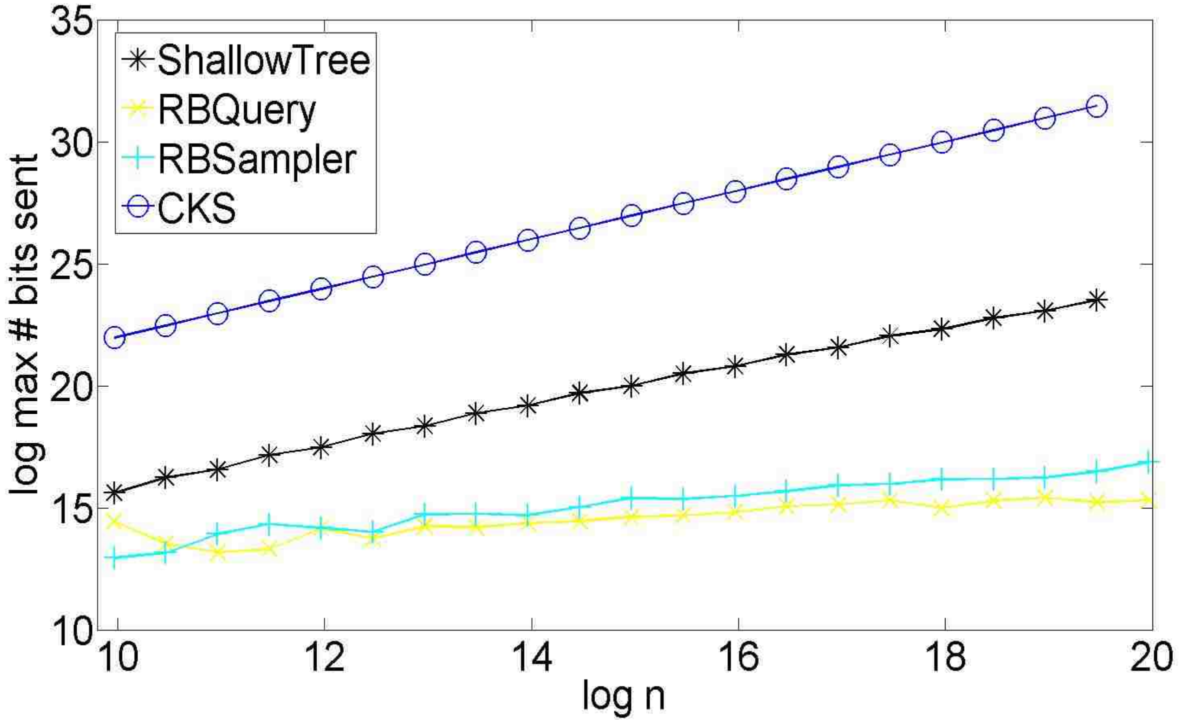


Figure 5.2: Top: Log of number of nodes vs. number of bits sent; Bottom: Log of number of nodes vs max bits sent by a node.

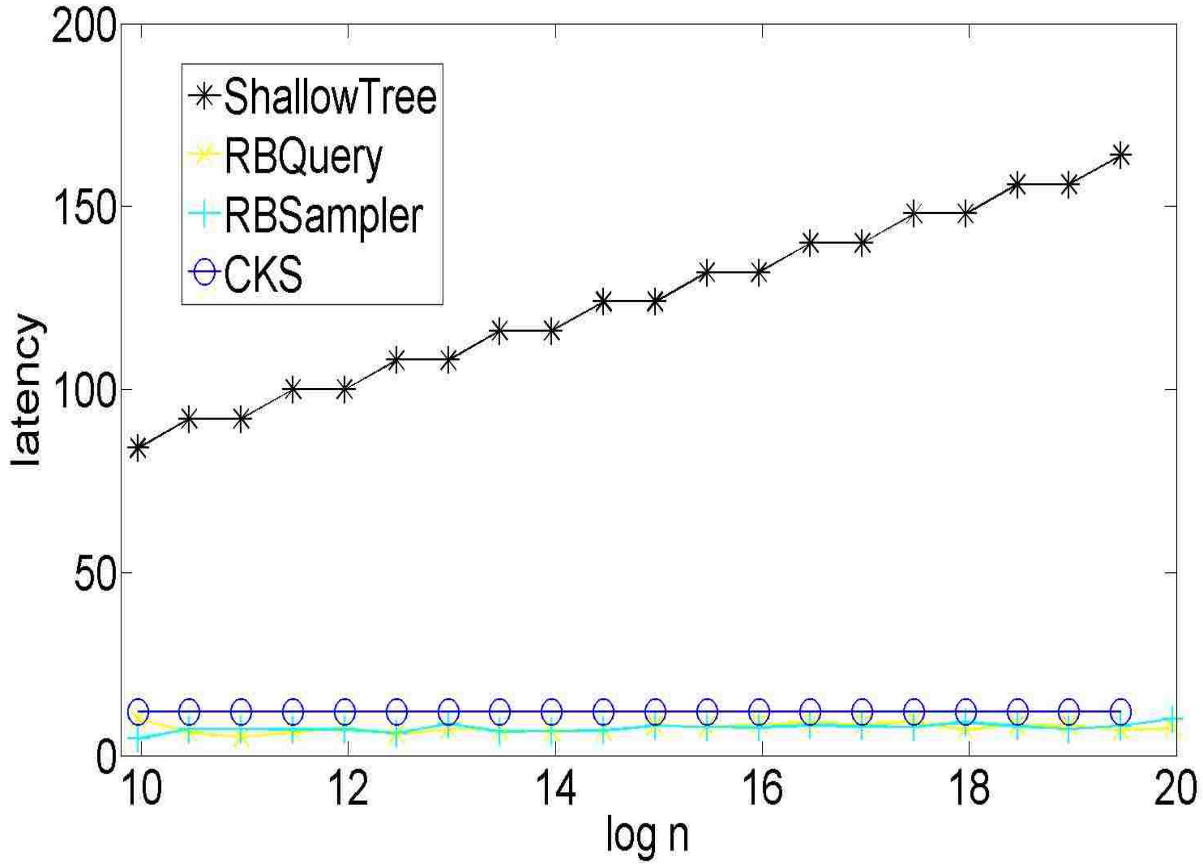


Figure 5.3: Algorithm Latency

5.4.1 Experimental setup

As in the previous chapter we used the OpenMP compiler directives in conjunction with the C++11 multi-threading primitives to speed up the simulation on a machine with 128G of memory and 48 cores. Because of the computational resources used by the machine we tracked only the total messages sent and the maximum number of messages sent by any processor. The size of the network simulated was between 1,000 and 1,024,000 processors.

For our simulations of RBQUERY, we selected a value of the constants as dictated by our analysis in section 5.3.1, $\epsilon_0 = 1/8$, $\epsilon > 1/6$, $C = 40$, we also made the size of the neighbors of a node in the algorithm to be $C \ln^2 n$. Increasing by a factor of $\ln n$ allows us to use a smaller C.

For our simulations of RBQUERY, we selected a value of the constants as dictated by our analysis in 5.3.2, $\epsilon_0 = 1/8$, $\epsilon > 1/6, C = 6, d = 3$, which implies that the size of the neighbors of a node in the algorithm is $C \ln^3 n$.

We compare our algorithms with the SHALLOWTREE and CKS algorithm from the previous chapter. Our results are averaged over 30 trials.

5.4.2 Results

Figure 5.1 (top) shows the logarithm of the total messages sent, We can see that our two algorithms RBQUERY and RBSAMPLER send less messages than the other algorithms when the network size is larger than 5,000 processors. Note that the RBQUERY algorithm sends slightly fewer messages than RBSAMPLER.

Figure 5.1 (bottom) show the maximum messages sent by a processor. RBQUERY and RBSAMPLER, have a maximum number of messages sent that is less than the other algorithms. This shows that the RBQUERY and RBSAMPLER algorithms seem to be load-balanced. RBQUERY is slightly better load-balanced than RBSAMPLER.

Figure 5.2 (top) shows the total bits sent. Again RBQUERY and RBSAMPLER send less bits than all the other algorithms for all network sizes we consider. The number of bits sent is dramatically less than that of the CKS algorithm since the size of a message in the CKS algorithm is about the same size as an RSA signature. Again the RBQUERY algorithm has a slight edge over the RBSAMPLER.

Figure 5.2 (bottom) shows the maximum bits sent by a processor. RBQUERY and RBSAMPLER again perform better than the other algorithms for all the network sizes we consider. RBQUERY algorithm sends less bits than RBSAMPLER.

Figure 5.3 shows the latency of the algorithms. The latency of our algorithms RBQUERY and RBSAMPLER, seem to be constant and about the same. The RBQUERY and RBSAMPLER algorithms seem to have slightly smaller latency than CKS.

Chapter 6

Conclusion and Future Work

We now briefly review the main contributions of this dissertation.

6.1 Conclusions

Figure, 6.1 shows all algorithms described in this dissertation. In Chapter 3, we described an algorithm (NATREE) for solving the Byzantine agreement problem using $\tilde{O}(\sqrt{n})$ average messages per node and $O(\log n)$ latency. This algorithm was robust against a non-adaptive adversary and tolerated $t < n(1/8 - \epsilon)$ faults for some positive ϵ. We simulated this algorithm and ran extensive experiments suggesting that for large networks, it requires significantly less bandwidth than the CKS algorithm from [16]. For all networks our algorithm required fewer total bits sent than the CKS algorithm, and for networks of size larger than about $65,000$, our algorithm also required fewer messages sent.

We showed in Chapter 4 that we can design scalable algorithms solving Byzantine agreement that are robust against an adaptive adversary. These algorithms send $\tilde{O}(\sqrt{n})$ messages per proces-

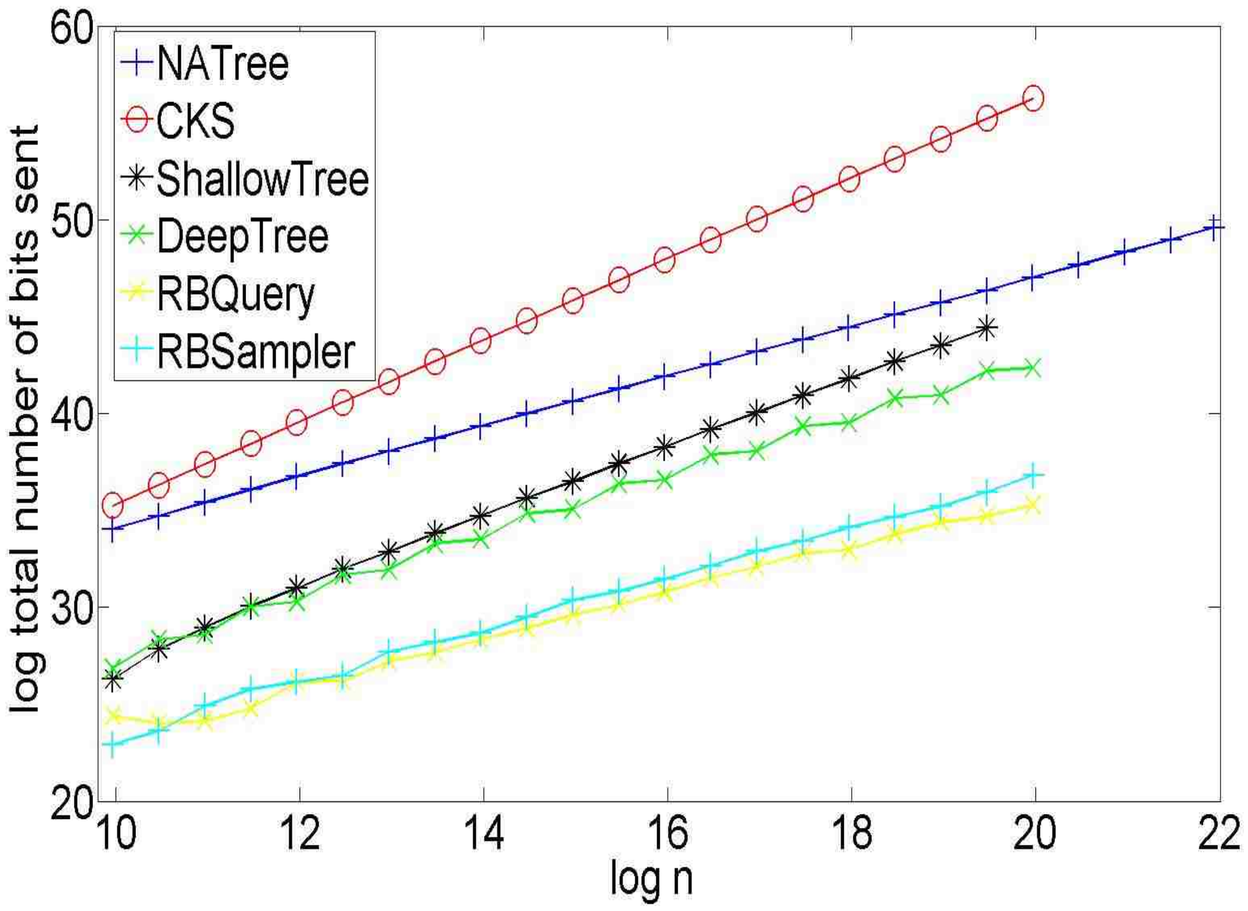


Figure 6.1: Total bits sent by all algorithms we described in this dissertation

sor and terminate in $O(\log n)$ rounds. The DEEPTREE algorithm tolerates $t < n(1/3-\epsilon)$ faults for any positive and ϵ, unfortunately the SHALLOWTREE algorithm only tolerates $\sqrt{n}$ adaptive faults, but tolerates $t < n(1/6-\epsilon)$ faults distributed uniformly at random for any positive ϵ. Unfortunately SHALLOWTREE has significant hot spots because of the tree structure: the processors that belong to the node at the top of the election graph send many more messages than the other processors. This is an undesirable property.

In Chapter 5 we showed that with the Random Beacon assumption, we can solve Byzantine agreement in $O(1)$ expected time, tolerating $t < n(1/3 - \epsilon)$ for any positive ϵ, adaptive Byzantine failures. RBQUERY sends $\tilde{O}(1)$ messages per processor if each bad node sends only $\tilde{O}(1)$ messages. RBSAMPLER always sends $\tilde{O}(1)$ messages per processor. One additional advantage of these algorithms is that they do not seem to develop large hot spots like the algorithms in Chapters 3 and 4. The algorithms have the best bandwidth and latency cost in practice anong all the algorithms we tested, for all network sizes we tested (between 1000 and 1,024,000). The algorithms are also very simple to implement. However, an obvious downside of the algorithms is the Random Beacon assumption and the additional synchronization constraints this assumption may imply.

6.2 Future work Byzantine agreement in general

We now discuss some areas for future work.

6.2.1 Improving Efficiency

The algorithm of Awerbuch and Scheideler [6] is a significant bottleneck for reducing bit cost in our algorithm. Reducing the message complexity of this algorithm even by a constant factor will reduce the total message complexity of our Byzantine Agreement algorithm significantly. The research question is a simple one can we design a distributed protocol, were a set of n processors can choose a set of numbers distributed uniformly at random, with message complexity less than $O(n^2)$? We are currently looking at ways to do this with secure multiparty computation.

6.2.2 Lower bounds

Some questions still remain to be answered about lower bounds on the Byzantine agreement problem some of which are:

1. We have already showed that we can solve Byzantine Agreement by sending $o(n)$ messages per processor. Does their exist an algorithm solving byzantine agreement in less than $\tilde{O}(\sqrt{n})$ messages per processor which tolerates $t < n(1/3 - \epsilon)$ adaptive faults for any positive ϵ?
2. Can we solve Byzantine agreement with $\tilde{O}(\sqrt{n})$ message complexity and with $O(1)$ latency?
3. If we relax the resilience of the algorithm to tolerating less than $t < \sqrt{n}$ adaptive failures, what does this buy us? What is the connection between message complexity (bandwidth) and resilience?

6.2.3 Asynchronous algorithm

An important open problem is designing scalable algorithms for the asynchronous communication model. We conjecture that there exists an algorithm in this model that requires each processor to sent $\tilde{O}(\sqrt{n})$ messages and has $\tilde{O}(1)$ latency.

6.2.4 Distributed Deployment

Another goal would be to implement each of our algorithms on a cluster of computers to do an actual distributed deployment of the algorithms on multiple processors. We would also like to run a comparison with several other algorithms to give a better picture of the performance of our algorithm. A challenge in this direction is finding a testbed platform that allows access to between 1000 and 1,000,000 nodes.

References

[1] Ittai Abraham, Danny Dolev, and Joseph Y. Halpern. An almost-surely terminating polynomial protocol for asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC '08, pages 405–414, New York, NY, USA, 2008. ACM.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 1–14, New York, NY, USA, 2002. ACM.

[3] Adnan Agbaria and Roy Friedman. A replication-and checkpoint-based approach for anomaly-based intrusion detection and recovery. In *Distributed Computing Systems Workshops, 2005. 25th IEEE International Conference on*, pages 137–143, 2005.

[4] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 45–58, New York, NY, USA, 2005. ACM.

[5] Yonatan Aumann and Michael O. Rabin. Information theoretically secure communication in the limited storage space model. In *In Advances in Cryptology, Proceedings of the 19th Annual International Cryptology Conference (CRYPTO*, pages 65–79, 1999.

[6] Baruch Awerbuch and Christian Scheideler. Robust random number generation for peer-to-peer systems. In *OPODIS*, pages 275–289, 2006.

[7] Jason Baker, Chris Bond, James Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, pages 223–234, 2011.

[8] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, pages 503–513, New York, NY, USA, 1990. ACM.

[9] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: a system for secure multiparty computation. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 257–266, New York, NY, USA, 2008. ACM.

[10] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for noncryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 1–10, New York, NY, USA, 1988. ACM.

[11] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *PODC '94: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 183–192, New York, NY, USA, 1994. ACM Press.

[12] Abhinav Bhatele and V. Laxmikant. An evaluative study on the effect of contention on message latencies in large supercomputers. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[13] Gabriel Bracha. An asynchronous [(n - 1)/3]-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pages 154–162, New York, NY, USA, 1984. ACM.

[14] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[15] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the Ninth ACM Conference on Computer and Communications Security (CCS)*, pages 88–97, 2002.

[16] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography (extended abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 123–132, New York, NY, USA, 2000. ACM.

[17] Christian Cachin and Ueli Maurer. Unconditional security against memory-bounded adversaries. In *In Advances in Cryptology CRYPTO 97, Lecture Notes in Computer Science*, pages 292–306. Springer-Verlag, 1997.

[18] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 42–51, New York, NY, USA, 1993. ACM.

[19] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[20] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[21] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[22] Chien-Fu Cheng, Shu-Ching Wang, and Tyne Liang. The anatomy study of server-initial agreement for general hierarchy wired/wireless networks. *Computer Standards & Interfaces*, 31(1):219 – 226, 2009.

[23] Jeremy Clark and Urs Hengartner. On the use of financial data as a random beacon. In *Proceedings of the 2010 international conference on Electronic voting technology/workshop on trustworthy elections*, EVT/WOTE'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[24] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2009.

[25] Allen Clement, Mirco Marchetti, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. Byzantine fault tolerance: the time is now. In *LADIS '08: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–4, New York, NY, USA, 2008. ACM.

[26] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *In Proceedings of Operating Systems Design and Implementation (OSDI)*, San Diego, CA, USA, 2005.

[27] Danny Dolev. The byzantine generals strike again. *J. Algorithms*, 3(1):14–30, 1982.

[28] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, 1985.

[29] C Dwork, D Peleg, N Pippenger, and E Upfal. Fault tolerance in networks of bounded degree. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 370–379, New York, NY, USA, 1986. ACM Press.

[30] Stefan Dziembowski and Ueli Maurer. Tight security proofs for the bounded-storage model. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, STOC '02, pages 341–350, New York, NY, USA, 2002. ACM.

[31] D. Eastlake 3rd. Publicly Verifiable Nomcom Random Selection. RFC 2777 (Informational), February 2000. Obsoleted by RFC 3797.

[32] D. Eastlake 3rd. Publicly Verifiable Nominations Committee (NomCom) Random Selection. RFC 3797 (Informational), June 2004.

[33] Uriel Feige. Noncryptographic selection protocols. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 142, Washington, DC, USA, 1999. IEEE Computer Society.

[34] Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM J. Comput.*, 26(4):873–933, 1997.

[35] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[36] Shafi Goldwasser, Elan Pavlov, and Vinod Vaikuntanathan. Fault-tolerant distributed computing in full-information networks. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 15–26, Washington, DC, USA, 2006. IEEE Computer Society.

[37] Ronen Gradwohl, Salil P. Vadhan, and David Zuckerman. Random selection with an adversarial majority. In *CRYPTO*, pages 409–426, 2006.

[38] Dan Holtby, Bruce M. Kapron, and Valerie King. Lower bound for scalable byzantine agreement. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 285–291, New York, NY, USA, 2006. ACM.

[39] Michael Isard. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41:60–67, April 2007.

[40] Bruce Kapron, David Kempe, Valerie King, Jared Saia, and Vishal Sanwalani. Fast asynchronous byzantine agreement and leader election with full information. In *SODA '08: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, New York, NY, USA, 2008. ACM Press.

[41] Anna Karlin and Andrew Chi-Chih Yao. Probabilistic lower bounds for byzantine agreement. Manuscript, 1986.

[42] Valerie King and Jared Saia. From almost everywhere to everywhere: Byzantine agreement with $\tilde{O}(n^{3/2})$ bits. In *To appear in Proceedings of DISC 2009:23rd International Symposium on Distributed Computing. Elche/Elx, Spain, September 23-25, 2009*, 2009.

[43] Valerie King and Jared Saia. Breaking the O(n^2) bit barrier: scalable byzantine agreement with an adaptive adversary. In *PODC*, pages 420–429. ACM, 2010.

[44] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 990–999, New York, NY, USA, 2006. ACM Press.

[45] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Towards secure and scalable computation in peer-to-peer networks. In *FOCS '06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 87–98, Washington, DC, USA, 2006. IEEE Computer Society.

[46] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, 2007.

[47] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35:190–201, November 2000.

[48] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[49] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[50] Hui Huang Lee, Ee chien Chang, and Mun Choon Chan. Pervasive random beacon in the internet for covert coordination. In *In: Information Hiding, 7th International Workshop, IH 2005 (LNCS 3727*, pages 53–61. Springer, 2005.

[51] Anna Lysyanskaya. Efficient threshold and proactive cryptography secure against the adaptive adversary (extended abstract).

[52] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distrib. Comput.*, 11(4):203–213, 1998.

[53] Ueli M. Maurer. Conditionally-perfect secrecy and a provably-secure randomized cipher. *Journal of Cryptology*, 5:53–66, 1992.

[54] R. J. McEliece and D. V. Sarwate. On sharing secrets and reed-solomon codes. *Commun. ACM*, 24(9):583–584, 1981.

[55] J.F. Meyer, D.G. Furchtgott, and L.T. Wu. Performability evaluation of the SIFT computer. *Computers, IEEE Transactions on*, 100(6):501–509, 2006.

[56] Hector Garcia Molina, Frank Pittelli, and Susan Davidson. Applications of byzantine agreement in database systems. *ACM Trans. Database Syst.*, 11(1):27–47, 1986.

[57] National Institutes of Standards and Technology. Federal information processing standards publication 180-3:secure hash standard. World Wide Web electronic publication, 2007.

[58] National Institutes of Standards and Technology. Federal information processing standards publication 186-3: Digital signature standard. World Wide Web electronic publication, 2007.

[59] O. Oluwasanmi, J. Saia, and V. King. An empirical study of a scalable byzantine agreement algorithm. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –13, April 2010.

[60] Arpita Patra, Ashish Choudhary, and Chandrasekharan Pandu Rangan. Simple and efficient asynchronous byzantine agreement with optimal resilience. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 92–101, New York, NY, USA, 2009. ACM.

[61] Arpita Patra and C. Pandu Rangan. Brief announcement: communication efficient asynchronous byzantine agreement. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 243–244, New York, NY, USA, 2010. ACM.

[62] M. Rabin. Randomized Byzantine generals. In *Proc. Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[63] Michael O. Rabin. Transaction protection by beacons. *Journal of Computer and System Sciences*, 27:256–267, 1983.

[64] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Awarded best student paper! - pond: The oceanstore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2003. USENIX Association.

[65] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, December 1990.

[66] Adi Shamir. How to share a secret. *Commun. ACM*, 22:612–613, November 1979.

[67] SINTRA. SINTRA-Distributed Trust on the Internet. http://www.zurich.ibm.com/security/dti/.

[68] Sam Toueg. Randomized byzantine agreements. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pages 163–178, New York, NY, USA, 1984. ACM.

[69] Marton Trencseni and Attila Gazso. Keyspace: A consistently replicated,highly-available key-value store. Published Online at http://scalien.com.

[70] Eli Upfal. Tolerating a linear number of faults in networks of bounded degree. *Inf. Comput.*, 115(2):312–320, 1994.

[71] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, PM Melliar-Smith, R.E. Shostak, and C.B. Weinstock. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, 1978.

[72] Andrew C. Yao. Protocols for secure computations (extended abstract). In *Proc. Symposium on Foundations of Computer Science*, 1982.

[73] W. Zhao. A byzantine fault tolerant distributed commit protocol. In *Dependable, Autonomic and Secure Computing, 2007. DASC 2007. Third IEEE International Symposium on*, pages 37–46. IEEE, 2007.

[74] David Zuckerman. Randomness-optimal oblivious sampling. In *Proceedings of the workshop on Randomized algorithms and computation*, pages 345–367, New York, NY, USA, 1997. John Wiley & Sons, Inc.

Appendix A

Algorithms and further technical proofs

A.1 Algorithms

A.1.1 Arrays and the election subroutine

Here we describe the array of random bits generated by each processor. Each processor generates a sequence of blocks, each except the last to be used for the elections in the nodes along the path to the root. The last block is only used to decide the final output bit, and contains only the bits needed to compute *a.e.BA-with-random-source*. To understand the blocks, we first describe Feige's election procedure, which is designed for an atomic broadcast model, and present a lemma about its performance.

Feige's election procedure*[33]: Each candidate announces a bin number in some small range. The good candidates choose a bin number selected uniformly at random in this range. The winners are the candidates which choose the bin picked by the fewest.*

Lemma A.1.1. *[33] Given a set of r numbers in the range $[1, .., numBin]$, let S be subset of these* *numbers which are generated independently at random by good processors, where the remaining numbers are chosen by an adversary which can view S before deciding on its values. For any constant c and sufficiently large n, if $|S|/numBin \geq (2c+2)\log^3 n$, then with probability at least $1 - 1/n^c$, the fraction of winners which are from S is at least $|S|/r - 1/\log n$.*

We show how to adapt Feige's election method to a set of processors which communicate by message-passsing.

The *input* to an election is an a.e. sequential (r, s) random sequence of blocks labelled $B_1, ..., B_r$. Each *block* B is a sequence of bits, beginning with an initial word (bin choice) $B(0)$ followed by r words $B(1), B(2), .., B(r)$. The number of bins is $numBins$ and each word has $\log(numBins)$ bits. After the election, the *output* of each processor is a set of winning indices W. Let $w = |W|$. Given $r \geq 2\log^4 n$, we set $numBins$ so that $w = r/numBins = \log^4 n$.

We now describe the election subroutine:

1. In parallel, for $i = 1, ..., r$, the processors run *a.e.BA-with-random-source* on the bin choice of each of the r candidate blocks. Round j of *a.e.BA-with-random-source* to determine i's bin choice is run using the i^{th} word of the j^{th} processor's block $B_j(i)$.
   Let $b_1, ..., b_r$ be the decided bin choices.

2. Let $min = i$ s.t. $|\{j \mid b_j = i\}|$ is minimal.
   $W \leftarrow \{j \mid b_j = min\}$.
   (If $|W| < r/numBins$ then W is augmented by adding the first $r/numBins - |W|$ indices that would otherwise be omitted.)

Then Feige's result can be extended to this context as follows (proof omitted due to space constraints)

Lemma A.1.2. *Let $r \geq 2\log^4 n$ and $s \geq (2c+2)r/\log n$. If the set of processors has a $2/3 + \epsilon$ fraction of good processors, then with probability $1 - 1/n^c$ the election results in agreement on a* *set of winners such that the fraction of winners from S is at least $|S|/r - 1/\log n$ and the winners are agreed to by a $1 - 1/\log n$ fraction of good processors. Choosing d so that $r \log numBins < \log^d n$, the election algorithm runs in time $O(r)$ and $O(r \log^d n)$ bits per processor.*

A.1.2 Algorithm and proofs for ALMOST-EVERYWHERE-BYZANTINE-AGREEMENT-WITH-RANDOM-SOURCE

We detail the algorithm and theorems from [43].

```
Set $vote \leftarrow b_i$; For s rounds do the following:
1: Send vote to all successors in G;
2: Collect votes from all predecessors in G;
3: $maj \leftarrow$ majority bit among votes received;
4: $fraction \leftarrow$ fraction of votes received for $maj$;
5: $coin \leftarrow$ left most bit from random beacon;
6: **if** $fraction \geq (1 - \epsilon_0)(2/3 + \epsilon/2)$ **then**
7:     $vote \leftarrow maj$
8: **else**
9:     $vote \leftarrow coin$
```

Algorithm A.1.1: A.E.BA-WITH-RANDOM-SOURCE

Theorem A.1.1. *[a.e.BA-with-random-source] Given a set of processors such that a $2/3 + \epsilon$ fraction are good and given a (s, t) a.e. random sequence of s bits, then for any fixed, positive integer d, a.e.BA-with-random-source runs in time $O(s)$ with bit complexity $O(s \log^d n)$ per processor. With probability at least $1 - 1/2^{t-1}$, all but a $1/\log^d n$ fraction of the good processors commit to the same vote b. Further, if all but a $1/\log n$ fraction of good processors have the same input bit b then b is the output bit.*

Corollary A.1.1. *Given a set of processors such that a $2/3 + \epsilon$ fraction are good and given a (s, t) a.e.random-source which generates s words of size w, there is an algorithm which runs in time*

$O(s)$ with bit complexity $O(sw \log^d n)$ per processor, such that with probability at least $w/2^{t-1}$, all but a $w/\log^d n$ fraction of the good processors output the same word. Moreover, if all but a $1/\log n$ fraction of good processors have the same input word b then b is the word output by all but a $w/\ log^d n$ fraction of the good processors.

A.1.3 Main protocol for a.e. BA

A.1.4 Almost-everywhere to everywhere algorithm

We now give the Almost-everywhere to everywhere(AE2E) algorithm from [43].

```
1. Generate Secrets and Send Up Shares:
   For all $i$ in parallel
      (a) Each processor $p_i$ generates an array of $\ell$ blocks $B_i$ and uses $secretshare$ to share its
          array with the $i^{th}$ level 1 node;
      (b) Each processor in the $i^{th}$ level 1 node uses $sendSecretUp$ to share its 1-share of $B_i$
          with its parent node and then erases its shares from memory.

2 Elect Winners at Root Node:
   Repeat for $\ell = 2$ to $\ell^* - 1$
      (a) Expose bin choices:
          For each processor in each node $C$ on level $\ell$:
          for $t = 1, ..., w$ and $i = 1, ..., q - 1$, let $B_{(i-1)w+t}$ be the $t^{th}$ array sent up from child $i$.
          ( If $\ell = 2$ then $w = 1$ )
          $W \leftarrow B_1 \| B_2 \| ... \| B_r$
          Let $F$ be the sequence of first blocks of the arrays of $W$, i.e., the $i^{th}$ array of $F$ is the
          first block of the $i^{th}$ array of $W$. Let $S$ be the sequence of the remaining blocks of each
          array of $W$.
          In parallel, for all candidates $i = 1, 2, .., r$
             i. $sendDown(F_i(1))$;
             ii. $sendOpen(F_i(1), \ell)$.
```

Algorithm A.1.2: *a.e.BA*

```
2 Elect Winners at Root Node(contd):

    (b) Agree on bin choices (contd):
        If $\ell < \ell^*$ then for rounds $i = 1, ..., r$
          i. Expose coin flips: Generate $r$ random words for the $i^{th}$ round of a.e.BA-with-random-source to decide each of $r$ bin choices.
             In parallel, for all contestants $j = 1, .., r$
              A. $sendDown(F_i(j))$; upon receiving all 1-shares, level 1 processors compute the secret bits $F_i(j))$;
              B. $sendOpen(F_i(j), \ell)$.
         ii. Run the $i^{th}$ round of a.e.BA-with-random-source in parallel to decide the bin choice of all contestants.
    (c) Send Shares of Winners Up to Next Level: Let $W$ be the winners of the election decided from the previous step (the lightest bin). Let $S'$ be the subsequence of $S$ from $W$; All processors in a node at level $\ell$ use $sendSecretUp(S')$ to send $S'$ to its parent node and erase $S'$ from memory.

3 Root Node Runs BA using Arrays of Winners:
  All processes in the single node on level $\ell^*$ run a.e.BA-with-random-source once using their initial inputs as inputs to the protocol (instead of bin choices) and the remaining block of each contestant. (Note that only one bit of this block is needed.)
  For rounds $i = 1, 2, ..., qw$ of a.e.BA-with-random-source,
    (a) $sendDown(F_i(1), i)$;
    (b) $sendOpen(F_i(1), \ell)$.
    (c) Use $F_i(1)$ as the coin for this round of a.e.BA-with-random-source.
```

Algorithm A.1.2: *a.e.BA* continued

Repeat s times:

1. Each processor p does the following in parallel:
   Randomly pick a set of $a\sqrt{n \log n}$ processors without replacement; for each of these processors j, randomly pick $i \in [1, \ldots, \sqrt{n}]$ (with replacement) and send a request label i to processor j.

2. Almost all good processors agree on the next number k in $[1, \ldots, \sqrt{n}]$ generated by some random source.

3. For each processor p, if p receives request label i from q and $i = k$ then if p has not received more than $a\sqrt{n}\log n$ such messages (it is not overloaded), p returns a message to q. 4. Let k_i be the number of messages returned to p by processors sent the request label i. Let i_{max} be an i such that $k_i \geq k_j$ for all j. If the same message m is returned by $(1/2 + 3\epsilon/8)a\,\mathrm{logn}$ processors which were sent the request label i_{max} then p decides m.

Algorithm A.1.3: Almost-everywhere to everywhere(AE2E) algorithm