

PROACTIVE AND REACTIVE APPROACHES FOR DYNAMIC WORKLOADS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Stavros Nikolaou

January 2017

© 2017 Stavros Nikolaou
ALL RIGHTS RESERVED

PROACTIVE AND REACTIVE APPROACHES FOR DYNAMIC WORKLOADS

Stavros Nikolaou, Ph.D.

Cornell University 2017

Many systems are designed to handle workloads with characteristics that are assumed to be static. Today's distributed systems, however, need to support a wide range of applications that incur dynamic workloads. As a result, the ability of a distributed system to adapt its operation to changing workloads is becoming an increasingly important property. There are two main types of approaches for supporting adaptivity: reactive approaches, which enable reconfiguration in response to a particular workload change, and proactive approaches, which continuously reconfigure the system in order to meet the demands of expected workload variations.

The type of approach a system can use depends highly on the available knowledge on the targeted workload. On the one end of the spectrum there are workloads whose changes can be accurately predicted, for example in the case of well-known periodic effects related to the access patterns of a particular service. Proactive approaches are a good candidate for such workloads since they can exploit prior knowledge of the workload's characteristics and drive the system to configurations that better suit the target workload. On the other end, there are workload changes that are very difficult to predict, e.g. attacks issued against the system in order to degrade some aspect of its operation. For unexpected workload changes, reactive approaches can be used to reconfigure the system when such changes are detected.

The focus of this dissertation is on designing and implementing distributed systems that support adaptivity. We present two main approaches, one proactive and one reactive, for dealing with two different kinds of dynamic workloads. The first describes proactive cache placement strategies for a cooperative cache built from individual client caches in an online social network or web service. We evaluate these strategies through simulations and compared with other common placement strategies under different workload scenarios. The second approach proposes a new asynchronous consensus protocol that we call *Turtle Consensus*. Turtle Consensus works by reactively reconfiguring the consensus strategy as well as the set of nodes upon which the strategy is executed in order to deal with denial-of-service attacks. We use the resulting protocol to implement a state machine replication protocol which is evaluated against various adversarial scenarios. The results suggest that Turtle Consensus can achieve great performance under benign scenarios and comparable performance while under attack.

This thesis presents actionable results from two ends of a spectrum of approaches to on-the-fly adaptation in distributed systems. While it is not the first to show examples of adaptive systems or argue its importance, our contributions significantly advance the state-of-the-art by pushing on boundaries at the ends of the spectrum of reactive and proactive approaches.

BIOGRAPHICAL SKETCH

Stavros Nikolaou was born in Patras of Greece in 1987. He received his Diploma in Computer Science from the University of Patras in June 2010. He then joined the Computer Technology Institute and Press "Diophantus" (Patras) as a research member on July 2010, where he stayed for one year. He started his Ph.D. in computer science at Cornell University in August 2011. He is engaged to the spectacular Dr. Tawny Nicole Cuykendall, who he met during his graduate studies.

Dedicated to my parents, fiancée, and close friends whose love and support have
got me this far.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor Robbert van Renesse, whose guidance has been invaluable during my years at Cornell. His dedication and persistence in helping me become a better computer scientist as well as his deep knowledge of the field have been a great inspiration to me. Working with him has been a great honor.

I would also like to extend my appreciation to the rest of my committee, Johannes Gehrke, Dexter Kozen, and Eva Tardos. Their advice has helped me navigate through the graduate program and their courses have stimulated my interest and deepened my appreciation for other areas of computer science.

I would like to thank my colleagues, labmates and friends, Karn Seth, Zhiyuan Teo, Sidharth Telang, Adith Swaminathan, Theodoros Gkountouvas, and Elisavet Kozyri for their friendship and our enlightening and stimulating discussions about research, life, the universe and everything. They provided insight, perspective and helpful feedback to my research endeavors.

Last but certainly not least, I would like to thank my family, Dionysios, Sofia, and Andreas who have constantly believed in me and have done everything in their power to help me be the best version of myself, and my fiancée Tawny for her patience, understanding, unending support as well as for bringing balance to my life during my years at Cornell.

This work is supported in part by DARPA grant FA8750-11-2-0256, AFOSR grant FA95550-11-1-0137, and grants by NSF, Facebook, Microsoft Corporation and Intel. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

TABLE OF CONTENTS

Biographical Sketch	iii
Dedication	iv
Acknowledgements	v
Table of Contents	vi
List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Proactive approaches for cooperative caching	2
1.2 Reactive Moving Target Defense for fault-tolerance solutions	3
1.3 List of Contributions	5
1.4 Roadmap of this dissertation	7
2 Background and related work	8
2.1 Caching in content delivery networks	9
2.1.1 Cooperative Caching	9
2.1.2 Cache placement	10
2.2 Reconfigurable consensus	11
2.2.1 Consensus protocols	11
2.2.2 Protocol switching	12
2.2.3 Denial-of-service attacks and moving target defense	14
2.2.4 Attack tolerance of existing fault-tolerance solutions	16
3 Proactive cache placement approaches for cooperative caching	18
3.1 Introduction	18
3.2 System Model	20
3.2.1 The Service and its Clients	21
3.2.2 Client-side Caches	23
3.2.3 Integrity and Privacy	24
3.3 Cache Placement Strategies	25
3.3.1 Baseline Approaches	25
3.3.2 Proactive Approaches	27
3.3.3 Space and Time Complexity	28
3.3.4 Optimal Variants	30
4 Evaluation of proactive cache placement schemes	31
4.1 Workload	31
4.2 Evaluation	33
4.2.1 Setup	34
4.2.2 Base case comparison	38
4.2.3 Optimal case comparisons	45
4.2.4 Replication factor	47
4.2.5 Churn	47

4.2.6	Neighborhood Access Probability	51
4.2.7	Results Summary	53
4.3	Concluding remarks	54
5	Turtle Consensus: A Case of Reactive Protocol Switching	55
5.1	Introduction	56
5.2	Background	58
5.2.1	System Model	58
5.2.2	Denial-of-Service	59
5.2.3	Consensus Protocols	60
5.3	Turtle Consensus	63
5.4	Implementation	74
5.5	Evaluation	79
5.5.1	Experiment Setup	79
5.5.2	Results	81
5.6	Concluding remarks	88
6	Moving Participants Turtle Consensus	89
6.1	Introduction	89
6.2	Model	91
6.2.1	Processes and communication	91
6.2.2	Adversary and attacks	92
6.2.3	Cryptographic primitives	94
6.2.4	Underlying consensus protocols	96
6.3	Moving Participants Turtle Consensus	99
6.3.1	Participants and participant sets	99
6.3.2	Configurations	100
6.3.3	Initialization and trusted dealer	101
6.3.4	Protocol description	104
6.3.5	Sketch of Correctness Proof	107
6.4	Extension to Byzantine Failures	116
6.5	Implementation	123
6.5.1	MPTC-based state machine replication	128
6.6	Evaluation	137
6.6.1	Setup	137
6.6.2	Results	140
6.7	Concluding Remarks	143
7	Conclusions	146
	Bibliography	149

LIST OF TABLES

3.1	Space and message complexity of the cache placement strategies. Here n is the number of clients and m the number of objects. . . .	29
4.1	Local hit ratio (LHR), global hit ratio (GHR) and average client bandwidth in KB/s (B/W) for each strategy without churn and optimal variants for 10,000 clients. The optimal variants of common neighbors and basic proactive approaches correspond to ProactiveOPTN and ProactiveOPT (from Figures 4.8a, 4.8b, and 4.9) respectively.	46
4.2	Local hit ratio (LHR), global hit ratio (GHR) and average client bandwidth in KB/s (B/W) for each strategy with moderate churn for 10,000 clients.	51
4.3	Average number of replicas per objects, evictions per client, and ratio of objects cached for each caching strategy for 10,000 clients.	53

LIST OF FIGURES

4.1	Average local hit ratio per client and global hit ratio of the collective cache (a), and average bandwidth per client (b) for the opportunistic approach with 10,000 clients as time passes.	36
4.2	Average local hit ratio per client (a) and local hit ratio running under a churn rate of 0.1 (b) for a varying number of clients.	38
4.3	Average node degree (a) and average number of replicas per key (b) for a varying number of clients.	39
4.4	Global hit ratio (a) and global hit ratio running under a churn rate of 0.1 (b) as the number of clients increases.	41
4.5	Ratio of objects cached at the end of each simulation over the full set of objects requested for a varying number of clients.	42
4.6	Average outgoing bandwidth per client (a) and outgoing bandwidth running under a churn rate of 0.1 (b) for a varying number of clients.	42
4.7	Average number of evictions per client for a varying number of clients.	43
4.8	Local hit ratio per client (a) and global hit ratio of the collective cache (b) of optimal algorithms as the number of clients increases.	43
4.9	Average outgoing bandwidth per client for optimal algorithms as the number of clients increases.	46
4.10	Average local hit ratio per client and global hit ratio of the collective cache (a) and average bandwidth per client (b) for the proactive approach running with 10,000 clients as the replication factor increases.	48
4.11	Average local hit ratio (a), global hit ratio (b) and average outgoing bandwidth (c) per client running under $NAP = 0.2$ for a varying number of clients.	49
5.1	Paxos throughput as a function of load and various attack scenarios.	82
5.2	TC-BenOr throughput as a function of load and various attack scenarios.	83
5.3	Turtle Consensus throughput (a) and latency(b) as a function of load alternating between Paxos and Ben-Or with and without attack.	85
5.4	Turtle Consensus throughput as a function of load running Paxos with different leader on each round.	87
6.1	A round of MPTC consensus.	106
6.2	Experiment topology.	138
6.3	Throughput as a function of load and various attack scenarios.	141
6.4	Latency as a function of load and various attack scenarios.	143

CHAPTER 1

INTRODUCTION

Distributed systems design is often driven by the characteristics of the workload that the task under consideration produces. Many of the solutions that appear in literature [102, 58, 62, 50, 35] exhibit great performance under the assumption that these workload properties are static throughout the system's operation but these systems' behavior may change significantly if these properties change over time. As online applications are becoming increasingly complicated and multifaceted the corresponding workloads they generate exhibit characteristics that vary over time. Such variations can be also caused by failures, changes in topology, connectivity and geographic distribution of the nodes upon which the system operates [17, 108, 48]. In addition, today's large scale distributed systems need to support a wide range of applications and tasks with different requirements and workload characteristics and operating under different configurations, which forms a highly dynamic environment for the system [101, 92, 14].

There are two main types of approaches that can be used to introduce adaptivity in a system. First, there are those approaches that assume knowledge of certain aspects of the workload and use this knowledge to proactively reconfigure the system in order to meet the demands of expected workload variations. Such approaches can be effective for dynamic workloads whose changes can be accurately predicted, for example workloads that demonstrate periodic shifts in access patterns or whose trends can be predicted by application-specific information. Proactive approaches can exploit this knowledge and drive the system to configurations that better serve the target workload.

Then, there are those cases in which the workload cannot be accurately modeled and predicted or where the cost for acquiring the knowledge required to proactively reconfigure the system is prohibitively high. Denial-of-service attacks constitute example workloads we cannot easily predict. In such scenarios reactive approaches can be used that enable the system to reconfigure certain aspects of its operation in response to detected workload irregularities.

This dissertation focuses on the problem of designing and implementing distributed systems that can adapt to dynamic workloads. It is divided in two main parts in which two different settings are explored with significantly different dynamic workload behavior. The first considers content delivery in Online Social Networks (OSNs) and represents a workload with predictable characteristics. The second considers denial-of-service (DoS) attacks in the context of fault-tolerant solutions and is characterized by unpredictable but significant workload variations. Within these contexts both proactive and reactive approaches are discussed and evaluated.

1.1 Proactive approaches for cooperative caching

The first part of this dissertation studies the cache placement problem in the context of a cooperative cache built from individual client caches in an OSN or web service. Motivated by the increasing popularity of various content-sharing web services, the increasing availability of information on the clients of such services and the advancement of technologies supported by client applications, cooperative caching poses an alternative to traditional content delivery systems by allowing the clients to cache and serve content to each other in a peer-to-peer (P2P) fashion. This approach offloads the service, naturally deals with flash-crowds, brings the

content closer to its consumers potentially improving client perceived latency and better distributes the bandwidth needed for content delivery across the network.

This dissertation proposes proactive cache placement strategies for a cooperative cache built on the clients of such services. These strategies employ information about clients' relationships that OSNs typically maintain. Based on these relationships and existing knowledge on workloads that these services experience, the proposed schemes proactively cache content on clients that are more likely to access it. One of those schemes is parameterized for tuning the degree of proactivity, thus controlling the scheme's tradeoff between benefits and costs. The schemes are evaluated against common and theoretically optimal placement strategies under different scenarios using simulations.

1.2 Reactive Moving Target Defense for fault-tolerance solutions

The second part of the dissertation considers fault-tolerance mechanisms and how state-of-the-art designs fail to deliver under the presence of well-coordinated DoS attacks. Such attacks can be viewed as unexpected variations in the workload perceived by the system. By studying consensus, one of the core building blocks of most fault-tolerance solutions, the dissertation investigates the design of consensus protocols needed for building attack-tolerant distributed systems. This line of work is motivated by the existing designs' focus on guaranteeing good performance under gracious executions where no failures or attacks occur but significantly weakening those guarantees when the workload becomes adversarial. The aim is to design a reconfigurable consensus protocol that can be used to imple-

ment attack-tolerant systems achieving good performance in both gracious and adversarial scenarios.

The resulting protocol presented in this dissertation is called *Turtle Consensus*, and it is inspired by previous work on protocol switching [68, 106, 47] as well as diversity [91] in time and space. Turtle Consensus is a round-based consensus protocol that changes the consensus strategy on-the-fly when it fails in making progress. It is a reactive approach to Moving Target Defense that leverages the diverse characteristics of the existing consensus protocols designs in order to achieve acceptable performance even against protocol-specific DoS attacks. Turtle Consensus' design is general enough to support a variety of settings by limiting the selection of strategies at each given protocol round. The implementation of Turtle Consensus is described and evaluated under different attack scenarios, which show promising results for dealing with particular DoS attacks.

To further strengthen this design, an extension to Turtle Consensus is also presented that add another degree of freedom to the protocol. This extension allows Turtle Consensus to switch between different sets of processes executing some consensus strategy across different rounds. The design uses existing cryptographic techniques to ensure that the reconfiguration of the protocol's execution characteristics cannot be predicted by even a strong adversary, capable of compromising a limited portion of the system. The design is also extended to byzantine failure environments where a bounded number of processes may behave arbitrarily.

1.3 List of Contributions

In this dissertation we make contributions in both the proactive and reactive sides of the distributed systems reconfiguration spectrum.

Our contributions on proactive adaptation are as follows:

- We propose two novel cache placement strategies that take advantage of known relationships between clients (for example, social links) and the workload on the service. Both schemes proactively create cached copies of content on clients that are more likely to access it based on those relationships;
- We evaluate our approaches and compare them with three common placement strategies. Our evaluation uses simulation on synthetically generated graphs and workloads that match certain characteristics of OSNs [10, 93]. Our findings suggest that we can substantially improve clients' hit ratios over common strategies at modest overheads;
- We also compare our proposed strategies with optimal placement schemes that assume full knowledge of the workload. We show that, under certain assumptions, we can get a near-optimal client hit ratio;
- We evaluate how our approaches work under churn. Our results show that these approaches are effective even under moderate churn.

On the reactive adaptation side we make the following contributions:

- We present Turtle Consensus, a round-based consensus protocol that embraces reactivity by dynamically changing the consensus strategy at each

round. Turtle Consensus is designed to achieve acceptable performance under DoS attacks that aim at exhausting the bandwidth of participants of the protocol.

- We present our experience with a prototype implementation of Turtle Consensus using two dissimilar underlying consensus protocols: Paxos and Ben-Or. We found that we can achieve both good performance in the absence of attacks and reasonable performance if the system is under attack while also heavily loaded by clients.
- We further extend the reconfiguration capabilities of Turtle Consensus by allowing it to additionally change the set of processes that participate in each underlying consensus protocol used. We call the resulting protocol Moving Participants Turtle Consensus (MPTC). While the sequence of configurations used in each round is predetermined by a trusted dealer, we use existing cryptographic techniques to ensure that next round's configuration can only be determined if sufficiently many processes collaborate in the current round.
- We additionally describe a byzantine fault-tolerant version of MPTC.
- We built a prototype implementation of crash-tolerant MPTC in which we used the same protocol across different rounds and kept changing the set of processes executing that protocol. Our evaluation suggests that we can achieve the performance offered by the most efficient consensus protocols even when the system is under attack.

1.4 Roadmap of this dissertation

This dissertation is organized as follows. Chapter 2 presents related work on cooperative caching, reconfigurable consensus, and denial of service attacks. Chapter 3 describes proactive cache placement strategies on a cooperative cache built from individual client caches in an online social network or web service. Chapter 4 presents our evaluation of the previous cache placement strategies that we conducted using synthetic workloads and simulation. Chapter 5 describes Turtle Consensus, a novel asynchronous consensus protocol that reactively changes consensus strategies on-the-fly in order to protect the system under denial-of- service attacks. It also presents our implementation and evaluation of our protocol. In Chapter 6, we describe an extension to the previous protocol that enables Turtle Consensus to not only change the protocol but also the set of processes that execute that protocol on-the-fly and in an unpredictable fashion using cryptographic techniques. Concluding remarks and future directions are discussed in Chapter 7.

CHAPTER 2

BACKGROUND AND RELATED WORK

There is a wide range of previous work on supporting adaptivity and reconfiguration in distributed systems that handle dynamic workloads for different settings. Such work spans load sharing policies in homogeneous [39] or heterogeneous [74] systems, in content distribution networks replication [69], routing protocols in mobile ad hoc networks [49, 87], as well as frameworks for general distributed systems focused on fault-tolerance [45, 27] and communication [107, 104]. This chapter focuses primarily on proactive and reactive reconfiguration and workload adaptation approaches related to cooperative caching and cache placement in content delivery networks (CDNs), as well as consensus protocols, protocol switching, attack tolerance, and denial- of-service attacks. The interested reader can find more related work on adaptive techniques and reconfigurable systems in topics not covered here like self- adaptive software [94], peer-to-peer systems [7, 70], and routing protocols [6, 5] for wireless sensor networks. These great surveys contain both proactive and reactive protocols used under various settings.

This chapter is structured as follows: In Section 2.1 we discuss previous reactive and proactive approaches in cooperative caching in the context of social networks as well as other distributed systems settings. In Section 2.2 we present previous work on primarily reactive approaches in the context of performance, fault-tolerance and DoS attacks.

2.1 Caching in content delivery networks

2.1.1 Cooperative Caching

Cooperative caching is a well-studied topic, particularly in the context of web caching [111, 109]. A representative system in this setting is the Shark [8] distributed file system, which uses sloppy DHTs to build locality-aware cooperative P2P caches between proxies. Another such system is Backslash [100], a collaborative P2P server/proxy caching for dealing with flash crowds using request redirection and URL rewriting. These systems focus on cooperative caches on proxy servers, hence they do not have the restrictions and dynamics of caches built on clients.

Our system model is based on the Maygh [113] cooperative client-side caching system. Maygh tries to reduce load on a web service by building a Content Delivery Network (CDN) on the clients' browsers. Another such system is Squirrel [54], a P2P web caching mechanism for geographically collocated clients (for example, within the same company) that uses a DHT for achieving scalability, self-organization, and churn tolerance. Our work focuses on Online Social Networks (OSNs), which are not a good candidate for Squirrel due to its requirement for geographically collocated clients. Also, neither of these approaches considered the problem of cache placement.

2.1.2 Cache placement

The cache placement problem has been studied in various settings. Several placement algorithms for web server replicas have been developed and evaluated in [85]. They use workload information, such as client latency and request rates, to make informed placement decisions. Our work performs cache placement by employing social link information and client preferences instead.

The work in [67] studies cache placement for web proxies under the assumption that the underlying network has a tree topology, modeling cache placement as a dynamic programming problem. [18] focuses on minimizing load on the network assuming a tree structure of limited depth and formulating a local greedy approach for finding a near-optimal solution. The work in [89] considers various replication and caching strategies within a simulated grid environment. The work views the grid as a tiered system and uses dynamic replication strategies to improve data access. Due to their restricted topology, these approaches do not generalize to clients of an OSN.

[103] proposes a cache placement strategy in OSNs similar to ours. This approach, called S-Clone, collocates replicas of data of neighboring clients with respect to the social graph of an OSN. Unlike our work, S-Clone replicates content on servers and not clients, avoiding the limitations of small cache capacities and churn. Also, S-Clone does not take workload information into account.

2.2 Reconfigurable consensus

2.2.1 Consensus protocols

Consensus is a critical component of many fault-tolerant services as well as a tool for reconfiguration of distributed systems. It lies at the core of replication protocols like state machine replication [63, 96] where it is used to achieve strong consistency guarantees by establishing a common order of input requests at all replicas. It is also a basic building block in various middleware configuration services like Google’s Chubby [20], Apache’s Zookeeper [57] and Microsoft’s Boxwood [71]. In these systems, consensus is used to implement distributed coordination services whose fault-tolerance rely on a small set of replicas and use variants of the Paxos [64] consensus protocol to keep these replicas consistent.

A wide range of crash-tolerant consensus protocols have been proposed in literature each optimized for a different setting and/or metric. Some were designed to handle datacenter-scale systems like [24], which describes how Paxos was used to implement a fault-tolerant database for the Chubby locking service, an instance of which lies in each Google’s datacenter. Others are focused on wide area deployments such as Mencius [72], which is a Paxos variant that employs multiple leaders, each of which is responsible for a different set of consensus instances and may reside at different datacenters. Another important differentiating aspect of consensus protocols is the number of proposers. On the one end of the spectrum there are protocols like Chandra and Toueg [26], Paxos [64], and many of its variants; they assume the existence of a special process, often called the leader or the active leader, that is responsible for making proposals to the rest of the processes. This coordinator-oriented approach has various advantages such as good

performance even under high contention and low communication complexity. On the other end, there are fully decentralized approaches [15, 86, 19, 9] where every process can make proposals and conflicts are resolved using randomization. Such decentralized approaches may be inferior in terms of performance in comparison to the leader-based ones but their performance is not affected by failures as much as the latter ones. For an excellent survey on consensus protocol see [41].

Our Turtle Consensus protocol can exploit these diverse characteristics of different consensus protocols by switching the protocol under execution during runtime.

2.2.2 Protocol switching

Our Turtle Consensus approach draws inspiration from various approaches to protocol switching for optimizing performance under different workloads and failure scenarios, however, none have applied it to counter Denial-of-Service attacks. [46] considers switching between primary-backup and state machine replication to achieve adaptive fault-tolerance. Similarly, [105, 106] explores protocol composition to build more flexible and efficient group communication systems and makes run-time decisions on the communication protocols used to optimize performance.

More recent work is the Abstract construction presented in [47]. In their work, they use the notion of abortable Byzantine Fault-Tolerant protocols (BFTs) that allow an existing BFT protocol to abort its execution and start another BFT protocol that can handle the condition that led the first one to abort. This way new BFT protocols can be built as a composition of existing ones, similar to how we use consensus protocols as black boxes and combining them across rounds. They separate the optimistic and failure cases to handle them as separate instances.

Instead, we are focused on the dynamic adaptation of the protocol to provide good performance in the normal case but also acceptable performance while under attack.

Based on the previous abortable BFT approach, [12] introduces ADAPT, which enables dynamic switching at runtime by selecting the next BFT protocol to run according to some objective function. Both previous approaches enable changing BFT protocols across different instances. Our Turtle Consensus approach does so at finer granularity by switching to different protocols across rounds, thus making use of any progress in existing consensus instances. In addition, our Moving Participants Turtle Consensus extension allows for further flexibility by changing the set of processes executing the underlying consensus protocols.

[95] introduces the idea of “slowness oracle” to create an adaptive coordinator-based consensus protocol which changes coordinator across round by selecting the most responsive process. This approach is dynamic but it relies on the existence of a special ordering module and can only change a single facet of the consensus mechanism. Turtle Consensus on the other hand can change all aspects of a consensus protocol during execution.

As an alternative to protocol switching, hybrid approaches like the one in [4] combine characteristics of different consensus protocols to strengthen some characteristic of consensus. The approach in [4] employs both the standard weakest failure detector $\diamond S$ and a random oracle to provide deterministic termination under no failures or failure detector inaccuracy, and probabilistic termination otherwise. The Turtle Consensus design allows for similar termination guarantees under certain implementations (for an example see Chapter 5, Section 5.4) but also enables

the combination of a larger set of protocols which can be applied to more diverse scenarios like denial-of-service attacks.

2.2.3 Denial-of-service attacks and moving target defense

Denial-of-service (DoS) attacks involve compromised hosts that saturate resources of the targeted system, typically the network, in order to prevent it from providing the service it is designed for. They are a common threat and have been studied in a variety of distributed systems settings, from internet web services [65] to sensor networks [112] and smart power grids [76]. The attacks considered in the work of Chapters 5 and 6 target the fault-tolerance component of a distributed system by attacking one of its core building blocks, its consensus mechanism. Since such fault-tolerance mechanisms are common in many distributed system we believe our methods to be applicable in most of the previous settings.

There is a variety of types of denial-of-service attacks. A coarse-grained classification with respect to the weakness exploited [75] is distinguishing between flooding or brute-force attacks and vulnerability or semantic attacks. In flooding attacks, the attacker either floods the network with a large amount of data in an effort to saturate the target's bandwidth or have the target process a large amount of data in order to saturate their CPU. TCP SYN, UDP, ICMP, DNS attacks and others have been extensively used [83, 37]. Vulnerability attacks try to exploit protocol or implementation-specific vulnerabilities to either saturate or incapacitate the target. Example attacks in this category try to exploit network device software or hardware issues like in the case of certain Cisco routers [30] in which the password checking routine could lead to a buffer overrun. Other attacks, like the Ping of Death [59], target operating systems implementations of certain

protocols. In the case of [59], sending ICMP echo requests of length greater than the maximum IP legal length could crash some of the targets. Some attacks target specific protocol features (e.g. DNS cache [34]) in which the attacker can force the target name server to cache inaccurate references. Finally, there are those attacks targeting specific aspects of applications running on the target in order to drain their resources. One example of such an attack is the finger bomb attack [3] in which the attacker can cause the finger routine to be recursively executed on the target, exhausting its resources. For more extensive listings and classifications of attacks and defenses see the following surveys: [75, 37].

The attacks considered in this dissertation are a combination of both the previous types since they are using UDP flooding to saturate the bandwidth of the target but also exploit protocol-specific knowledge to considerably degrade the performance of the fault-tolerance mechanism. There are many more attacks in the literature.

Moving target defenses have often been used as response to DoS and Distributed DoS (DDoS) attacks. [44] proposes changing the IP address of the target node for dealing with local IP-based DoS attacks. More recently in [55], Software-Defined Networking (SDN) has been used to implement moving target defense approaches like “random host mutation” in which, similarly to [44], the controller periodically alters the virtual IP addresses of hosts to hide the real IP addresses from an intruder. Our Moving Participants Turtle Consensus approach (Chapter 6) resembles more the “proactive server roaming” approach in [60]. That is an adaptive approach in which the active server proactively switches servers from an existing pool in order to deal with unpredictable and undetectable attacks. Their approach ensures that only legitimate clients can track the moving server. Like in the case of

our MPTC protocol, proactive server roaming performs gracefully during attacks. However, it imposes significant overhead in attack-free scenarios, which is not the case for MPTC since we only reactively change configurations. For a more in-depth description of the challenges, achievements, and future directions on the topic of moving target defense approaches see [56].

2.2.4 Attack tolerance of existing fault-tolerance solutions

While others have targeted good performance for the common case (for example, [84, 61]), our focus is on performance under Denial-of-Service. In that sense, our work is inspired by observations like those made in [31] and [99] regarding the robustness of state-of-the-art fault-tolerant state machine replication protocols. [99] showed that a range of popular BFTs such as [22, 33, 61] are sensitive to network conditions and that one-size-fits-all BFTs are hard to design. This strengthens our case for an adaptive approach that can employ the wealth of existing protocols to deal with the peculiarities of an adversary, whether the adversary is a rogue client or the network itself.

The authors of [31] show that even a single malicious client can render the system unavailable by carefully crafting a series of requests. They propose the Aardvark protocol, based on a set of insightful amendments to the well-known PBFT protocol [22]. These amendments concern signed requests, point-to-point communication, and regular view changes, and render Aardvark more robust to misbehavior of both clients and servers. Our work shares similar goals, though our solution is to utilize a set of existing protocols and combining their strengths. Also, Aardvark’s solution to bandwidth attacks relies on additional Network Interface Cards.

Our work can be also viewed through the lens of protocol transformation such as [51], where the authors turn a crash-tolerant protocol into a Byzantine-tolerant one. Our protocol can be categorized as such a transformation because it attempts to transform consensus protocols susceptible to DoS attack to one that can gracefully handle that attack.

CHAPTER 3
**PROACTIVE CACHE PLACEMENT APPROACHES FOR
COOPERATIVE CACHING**

This chapter investigates cache placement on a cooperative cache built from individual client caches in an online social network or web service. We use a service that maintains a mapping between content and the clients that cache it, and propose cache placement schemes that leverage relationships between clients (for example, social links) and workload statistics, proactively placing content on clients that are likely to access it.

3.1 Introduction

The increasing popularity of today’s content-sharing web services such as Online Social Networks (OSNs), photo-sharing websites, and video-on-demand systems is leading to vast amounts of generated content. Services need to either rely on Content Delivery Networks (CDNs) and cloud storage to meet the increasing demand, or build, deploy, and manage their own content delivery systems. Both approaches are expensive and thus are often only accessible to large companies.

An alternative is to deploy cooperative caching [54, 113], where clients of the service act as caches and serve content to each other in a peer-to-peer (P2P) fashion. Cooperative caching has multiple benefits: First, it offloads the service, thus mitigating the cost of content delivery. Secondly, it helps with flash crowds [100], since a flash crowd will provide the service with a proportionally large cache capacity to deal with the spike in demand. Thirdly, cooperative caching schemes can substantially reduce access latency since content resides close to the clients

themselves. Finally, it improves network resource utilization by distributing bandwidth usage to many client-to-client connections, offloading the service’s backbone network.

Cooperative caching has been around for some time, but it has recently gained renewed interest for various reasons. First, technology has become available that enables easy deployment of such schemes. Maygh [113] is an example system that leverages non-intrusive technologies such as Flash and WebRTC for browser-to-browser communication to show the feasibility and benefits of cooperative caching for today’s services. Secondly, there is increasing availability of relationship information between clients such as social links, users subscriptions to other users’ updates, etc. Such information can provide the service with better insights into access patterns, which can be leveraged for efficient cache placement strategies.

This work focuses on how to better organize the client caches in a cooperative cache setting and studies the cache placement problem for a cooperative cache built on the clients of an OSN or similar content-sharing services. Compared to our previous work [80], we have developed improved strategies, address the effect of client churn, use more realistic workloads to evaluate our strategies, and consider privacy issues that arise in cooperative caching.

We consider a system model similar to that of Maygh. Each client maintains a cache of limited capacity for caching content and can communicate with other clients. The service has a growing corpus of objects that can be requested by clients and maintains an approximate mapping between objects and the clients that cache those objects.

Because strategically placing cached content can significantly affect performance and overhead in such a cooperative environment [88], we investigate how cache placement affects hit ratio and load perceived by the service and the clients. We make the following contributions:

- We propose two novel cache placement strategies that take advantage of known relationships between clients (for example, social links) and the workload on the service. Both schemes proactively create cached copies of content on clients that are more likely to access it based on those relationships. One of these strategies allows tuning proactivity, thus controlling the approach’s tradeoff between benefits and costs;
- We evaluate our approaches and compare them with three common placement strategies. We also compare them with optimal placement schemes that assume full knowledge of the workload. Our evaluation uses simulation on synthetically generated graphs and workloads that match certain characteristics of OSNs [10, 93];
- We evaluate how our approaches work under churn.

Our findings suggest that we can substantially improve clients’ hit ratios over common strategies at modest overheads. We show that, under certain assumptions, we can get a near-optimal client hit ratio.

3.2 System Model

In this section, we describe the model we assume for our cooperative cache. The model consists of two main components: a service and the set of its clients. We

discuss these components and their interactions, and we consider security and privacy implications of cooperative caching.

3.2.1 The Service and its Clients

The main role of the service is twofold: First, in response to a client request, it either serves the contents of an object, or the location where the object is cached. Secondly, it determines which clients should cache the objects. We assume that objects are immutable; mutable objects are modelled as a sequence of versions, each being an immutable object. Each object has an owner—one of the clients of the service. We identify objects via *keys* that the service maintains in a key-value store along with the corresponding immutable content, the owner, and the set of clients that have recently cached the objects. Recency is here considered with respect to some time window determined by the service during which a client has been directed to cache an object. These directives are described in greater detail later in this section. The service tracks which clients are online by monitoring their most recent activity with respect to this time window.

Since we are focusing on OSNs, we assume that the service maintains a *client relationship graph* which encodes client subscriptions to the objects of a particular owner. In this graph, nodes represent clients of the service and edges represent client subscriptions. Such a graph can effectively model an OSN where client subscriptions represent mutual interest in the objects published by the corresponding clients. We assume the service either has access to such a graph or can construct it by enabling its clients to subscribe to each other’s objects. The exact details of how the service obtains the client relationship graph are outside the scope of this work. Our workload model is based on [16]: objects are accessed with a

heavy-tailed distribution. The client accessing such an object is selected as follows: with probability NAP (*Neighborhood Access Probability*) the client comes from the *neighborhood* of the owner of the object, that is, from either the owner itself or one of its neighbors in the client relationship graph. Otherwise (with probability $1 - NAP$), the client is selected uniformly at random from all clients excluding the ones in the neighbourhood of the owner.

The service interacts with the clients using what we call *cache directives*. A cache directive is a message that the service sends to a client either as a *reply* to a client’s request or as an *unsolicited cache directive* and has the following format: (ID, C, D, L) . ID is the identifier of an object (this can be a collision-resistant hash of its content), C is a boolean used to indicate whether the receiving client should cache the object or not, D is the content of the object (can be \perp to indicate that the content is not included in the message), and L is a set of cache locations, that is, a set of clients.

If a cache directive is a reply to a client’s request then ID holds the identifier of the object requested. If $D \neq \perp$ then the service is providing the object’s content to the client and $L = \emptyset$. If $C = \mathbf{true}$ then the service directs the client to store (ID, D) in the the client’s cache. Conversely, if $C = \mathbf{false}$, the client should not cache the object. If $D = \perp$ then the content is not provided in the reply and the receiving client needs to fetch it from one of the locations in L , and thus $L \neq \emptyset$. To do so, the client will issue a *side-load* request to locations in L . We describe side-load requests in Section 3.2.2.

If a cache directive is an unsolicited one there are two cases: First, the directive is of the form, $(ID, \mathbf{false}, \perp, \emptyset)$, in which case the receiving client must evict object ID from its cache if it is there. Second, the directive is of the form $(ID, \mathbf{true}, \perp,$

L), where $L \neq \emptyset$ and the receiving client must fetch object ID from one of the locations in L and cache it even though that client did not explicitly requested it. When the service issues an unsolicited cache directive of the second form, we say that the service *pushes* object ID to the receiving client.

3.2.2 Client-side Caches

Each client is equipped with a cache of limited capacity that it can use for serving client requests. A client may evict an object from the cache either when it receives an eviction directive or when it has to cache an object and its cache is full. In the latter case, the client decides which objects to evict using some eviction policy such as Least Recently Used (LRU), Least Frequently Used (LFU), or Adaptive Replacement Cache (ARC) [73].

A client can interact with the service and other clients using three types of messages: `request(ID)`, `side-load(ID)` and `cache-update(ID, U)`. When a client requires access to an object, ID , that does not reside in its cache, the client sends a `request(ID)` message to the service. The service responds with a reply cache directive like those described in Section 3.2.1. If the response has a non-empty location list, the requesting client sends `side-load(ID)` messages to each of the clients in the location list, starting with the first one. A client receiving a side-load request for some object, ID , checks its cache and if ID is there, responds with the content of the object; otherwise it responds with an error. If the requesting client receives an error or times out while waiting for a response from a location, it sends a `cache-update(ID, U)` message to the service. The U component of the previous message is a list containing the clients that did not respond in time with the content of the object requested. If the service receives such a cache-update message,

it updates its meta-data regarding the clients caching object ID . If none of the clients on the location list provided by the service is able to serve the object, the client obtains the object from the service directly by sending another `request(ID)` message. Once the requesting client obtains the content of object ID , it executes the cache directive provided by the service according to C 's value.

3.2.3 Integrity and Privacy

Cooperative caching has significant security concerns. As clients are serving objects, malicious clients can modify them. This, however, is easy to prevent with digital signatures [90].

A more difficult problem concerns the privacy of clients. If the server directs client c to obtain an object from client c' , then c and c' learn something about one another, and in particular may learn something about one another's interests. In a previous incarnation of our system [80], it was indeed trivial to learn this. We have made this significantly harder in our new approaches.

The main idea is the following: when a client c' receives a request for an object from another client c , c' cannot know if c requested the object from the service or whether the service sent c an unsolicited directive to cache the object. Vice versa, c cannot know if c' directly requested the object in its cache either. This gives both c and c' a level of *plausible deniability*. However, c' still learns that c is in the neighborhood of a client that requested the object. In order to increase the amount of plausible deniability, the service can sometimes select random clients to cache objects, similar to what was suggested for Maygh [113].

3.3 Cache Placement Strategies

In this section, we present various cache placement algorithms. The first three are intended for baseline comparisons. Next we present the proactive algorithms we have designed, leveraging social connections in order to improve individual client hit ratios. Then we present optimal versions of each algorithm, assuming knowledge of the future.

3.3.1 Baseline Approaches

Opportunistic Approach

The *opportunistic* scheme is a commonly used approach for CDNs and web caches. Maygh, BitTorrent, and Gnutella all use a variant of this approach. Upon receipt of a request, the service checks to see if there are clients caching the object. If so, the service responds with the locations of those clients. If not, the service provides the object itself. In either case, the service directs the client to cache the object ($C = \text{true}$), using LRU eviction if its cache is full.

Minimalistic Approach

The objective of the *minimalistic approach* is to minimize load on the server. To do so, it tries to keep at most one copy of an object in the collective cache of the clients. The scheme works as follows: When the service receives a request for an object, the service checks whether the object was recently cached by another client. If so, the service provides the requesting client with a singleton set L containing

the client that has recently cached the object and sets directive C to **false** (*that is*, it requests the client not to cache the object). Otherwise, the service sends a directive to the client containing the content and sets C to **true**. In that case the client caches the object. If its cache is full, then LRU replacement applies.

Minimalistic*

The minimalistic approach has two main problems. First, a client that has a copy of a highly popular object becomes a hotspot since it must serve all requests for the object. Second, in case of churn many client-to-client loads may fail. Minimalistic* is a variant that only tries to minimize the number of copies held for unpopular objects.

When the service receives a request for an object ID from a client c , it first checks to see if there is a client c' that has recently loaded the object. If not, the service returns the object to c with the directive to cache it. If there is a client c' , the service sends to c a directive $(ID, \mathbf{true}, \perp, \{c'\})$, directing c to load the object from c' and add the object to its cache. Upon receipt, c informs the service. If the popularity of ID is below a configured threshold, then the service directs c' to evict the object. The service approximates the popularity of an object by the fraction of requests issued for that object in a configured window of time.

For small thresholds, the algorithm approximates the opportunistic approach, because it is more likely that a copy of the requested object will be created on each request (for threshold value 0 they become identical). For large thresholds, the algorithm approximates the minimalistic algorithm.

3.3.2 Proactive Approaches

Basic Proactive Approach

The basic *proactive* algorithm leverages the client relationship graph by proactively pushing content to the neighborhood of the content owner. We here define the *neighborhood* of the content owner, or simply client, as the set containing the client itself as well as its neighbors in the client relationship graph.

When the service receives a request, it responds with either the object or a list of locations, L , as in the opportunistic scheme. The response is always a caching directive, that is, $C = \mathbf{true}$. In addition, the service selects the $\lceil r \times |N_{owner}| \rceil$ most recently active clients from the owner’s neighborhood, where $r \in (0, 1]$, the *replication factor*, is a parameter of the algorithm and N_{owner} is the set of neighbors of *owner*. The service issues a directive to each of them to side-load and cache the object, excluding any clients that appear off-line or have recently cached the object already. Clients that do not have enough space left in their cache use LRU eviction.

Common Neighbors Proactive

We also propose a variant of the previous approach that we call the *common neighbors proactive* scheme. With this variant, the subset of the owner’s neighborhood to which the object is pushed is the intersection of the owner’s neighborhood and the neighborhood of the client requesting the object.

3.3.3 Space and Time Complexity

The space complexity of the previous approaches is the state required by the service to keep track of online clients and their cached objects. All approaches require this meta-data and thus the space complexity for all approaches is $O(n \cdot m)$ where n is the number of clients of the system and m the total number of objects. Given an object identifier, the service needs to provide the set of clients that cache it. It must, therefore, keep a mapping from objects to caching clients. We assume that $m \geq n$, which is reasonable for social networks since in most social networks users typically need an account to access data on the service and thus at least a profile object must exist per user. Notice that due to the previous assumption, the additional client relationship graph required by the proactive approaches—which would typically require $O(n^2)$ space—is included in the previous bound. The size of the previous state increases substantially as the number of clients and objects increases, and thus additional measures need to be taken for keeping the state size manageable in popular social networks. Such measures may include garbage collection of old and/or unpopular objects and sharding of the service state across multiple machines, for example, by employing consistent hashing, geo-spatial information etc.

We consider time complexity with respect to the number of messages that need to be sent until the client receives the object it requested. We briefly describe the worst case analysis of a client’s request for each strategy. Each approach requires sending a message to the service to get a location of the cache, if it exists, or the object content otherwise. If a location is known, another message needs to be sent to that cache. From this point forward each approach may differ on the number of messages sent. In the minimalistic approach, a request will always be served

after at most three messages. The first two messages are as above. The last one is sent to the service for retrieving the object when the provided location is no longer available, either due to time-out, failure, or due to the object no longer being cached at the location. The remaining approaches are similar with respect to the number of messages. This number is bound by the number of locations provided by the service, that is, $O(n)$. Notice that, by our model description, each time a client detects a cache location unable to provide the object, it has to inform the service via another message. This communication, however, is not crucial for obtaining the object and thus can be performed in the background. The previous bound can be very impractical and thus, in a real deployment, we could employ some optimizations for reducing the number of messages sent. Such optimizations might include trimming the service response to a fixed number of locations and contacting multiple locations in parallel, at an additional bandwidth cost.

Table 3.1 summarizes the previous discussion.

Algorithms	Space	#Messages
Minimalistic	$O(n*m)$	3
All others	$O(n*m)$	$O(n)$

Table 3.1: Space and message complexity of the cache placement strategies. Here n is the number of clients and m the number of objects.

The previous discussion suggests that minimalistic is the best algorithm since it performs better with respect to message complexity and equally well with respect to the memory needed. In practical systems however, client specific metrics like latency perceived and bandwidth spent to support the collaborative cache that offloads the service are more important. In addition, the cache described in Section 3.2 needs to gracefully handle the churn that is inherent in the peer-to-peer nature of the system. As we later see in the evaluation of the approaches (Section 4.2),

the minimalistic strategies do not handle churn well. Finally, we note that all approaches scale similarly, though the proactive strategies do require additional effort for managing large scale client interaction graphs.

3.3.4 Optimal Variants

For comparison purposes, we have also implemented optimal variants of all previous algorithms. These variants assume knowledge of the future, which is provided in the form of an oracle that knows the trace of requests a priori.

For the minimalistic and opportunistic approaches, the oracle is used during the eviction process when the cache of the clients is full. In other words, clients implement Belady’s optimal algorithm for cache replacement [13].

For the proactive approaches, the oracle is used for optimal eviction as well as for optimal pushing. The optimal proactive schemes push copies only to those clients that will be requesting the object in the future.

There are two flavors of the optimal proactive algorithm. The first one, which we call *globally optimal proactive*, has the client push the object to all clients that will request it in the future (and potentially outside of the client’s neighborhood), excluding those that already cache it. The second flavor is called *neighborhood only optimal proactive* and works as the previous one with the restriction that copies are pushed only to clients that will request the object and are in the neighborhood of the owner.

CHAPTER 4

EVALUATION OF PROACTIVE CACHE PLACEMENT SCHEMES

In this chapter, we evaluate the efficacy of the cache placement strategies described in Chapter 3 through simulation. We compare our schemes against commonly used cache placement algorithms as well as optimal placement. We synthesize a workload to match characteristics of online social networks. Simulation results of our proposed caching schemes impose moderate network overhead and show considerable improvement to the client’s cache hit ratio, even under churn.

4.1 Workload

In this section, we describe how we generate the workload used in our evaluation (Section 4.2). Our main focus is to capture workload characteristics of realistic OSNs where clients may share and subscribe to each other’s content.

One of the workload aspects we are interested in is the corpus of objects requested. We observe that the corpus of objects in OSNs continuously grows as time passes (for example, due to photos and posts uploaded by users). As a consequence, the relative popularity of objects decreases over time: old content becomes stale and unpopular, whereas new content gets more attention.

To capture these observations, we assume a corpus represented as a list of keys ordered by the objects’ popularities. We insert new objects in this corpus at a rate of approximately 1/30th of the aggregate request rate. We consider key insertions to play the role of writes and all clients’ requests to be the reads to the service. This read-heavy workload is commonly observed in OSNs and approximates Facebook’s key-value store [10] with respect to the read/write ratio.

When a new object is added to the corpus, its popularity (that is, its rank in the corpus list) is picked from a Zipfian distribution with skew parameter α . Once the object’s rank is decided, its respective key is inserted in the corpus, increasing all equal or higher ranks by one. For each new object, an owner is selected uniformly at random from the set of clients. The key and content sizes of the newly inserted object are selected from the distributions provided by [10], reflecting Facebook’s key-value store and matching closely the object size distribution at Facebook [52]. Each time a new request is issued, we select the object according to a Zipfian distribution with skew parameter α .

According to the model above, the popularity of objects decreases as new objects are added in the corpus. We name this model the *shifting popularity model* (SP). This model was found to closely correspond to measurements performed on Facebook’s photo corpus [52]. The initial size of the corpus determines the speed at which the popularity of objects change. We revisit this issue in Section 4.2.

The time between client requests follows a Generalized Pareto distribution [10]. Request inter-arrival time is concentrated around $\sim 500\mu s$.

The last component of our workload generator is concerned with the client relationship graph that is used for selecting the client that issues a request for a given object. The specifics of the selection have been described in Section 3.2.1. For this work, we assume that the client relationship graph is static. This assumption simplifies cache location selection and generation of cache directives.

We leverage a graph model to synthetically generate graphs of any given size that share common graph characteristics such as node degree distribution, clustering coefficient, and others, with real social networks. There is a large number of

graph models for social networks [66, 78, 23]. In this chapter, we chose the modified *nearest neighbor graph model* [93], $G(n, u, k)$, because this model can generate graphs that approximate real social graphs best. The generation proceeds in steps. The model takes three parameters: 1) the total number of nodes, n , 2) a probability, u , that determines at each step if a new node is added or if a pair of 2-hop neighbors are connected, and 3) the number of node pairs, k , that are connected on a node addition. We derived the parameter values by fitting the nearest neighbor model to a set of real world graphs found in [1]. Our objective function for the fitting process takes into account the similarity of the generated graphs to the real ones with respect to node degree distribution and clustering coefficient.

Our request generation procedure takes a client relationship graph G , the NAP probability, and the skew parameter α as input and is described in Algorithm 1.

Algorithm 1 Request generation algorithm

```

1: procedure GENERATE REQUEST( $G, \alpha, NAP$ )
2:   select a key  $k$  according to the SP model with parameter  $\alpha$ .
3:   if  $p \leq NAP$  for  $p \in [0, 1]$  chosen uniformly at random then
4:     select the client  $v$  issuing the request uniformly at random
5:     from the object's owner and its neighbors.
6:   else
7:     select client  $v$  uniformly at random from all clients
8:     excluding the owner and its neighbors.
9:   end if
10:  return  $(v, k)$ 
11: end procedure

```

4.2 Evaluation

In this section, we compare the cache placement strategies described in Section 3.3. We assess key cache performance metrics through simulation on synthetically generated workloads as described in Section 4.1. There is a trade-off between

the fidelity of a simulation and the scale at which we can run such a simulation. In order to run the simulations in a reasonable amount of time, the number of clients and the number of objects in the corpus in our experiments are significantly smaller than in a popular social networking site. A large service will still require sharding and multi-level caching such as used at Facebook [52], but this section will demonstrate that medium-scale services can significantly benefit from proactive cooperative caching. Existing scaling techniques like those presented in [113] may be used for porting the ideas illustrated here to systems of larger scale.

4.2.1 Setup

In our experiments, we vary the number of clients n from 1 to 10,000 and keep the cache capacity per client, c , constant at 5MB. Many browsers, particularly on mobile devices, impose a 5MB limit on their HTML5 cache sizes [2]. We did experiment with larger cache sizes and found similar trends as reported below. The skew parameter of the Zipf distribution, α , is kept constant at 1.1. We obtained this parameter value from [43], where a similar decreasing popularity mechanism is described and found to approximate a popular social network. Experimentation with larger skews also result in similar trends.

For a range of n , we generate a client relationship graph G according to the nearest neighbor model (Section 4.1) with parameters $u = 0.96$ and $k = 1$. Then for each graph G , we generate a workload of 20 million requests using Algorithm 1 with NAP fixed to 0.8. The NAP value comes from [16] and is consistent with click stream behavior observed in social networks such as Orkut, LinkedIn, and others. (In Section 4.2.6 we investigate the effect of using a small NAP .) The size of the

trace corresponds to 150 minutes of execution using the request rate discussed in Section 4.1.

Even though our workload implies the same number of requests for different numbers of clients, the results do not change. This is because after the warm up phase, and given the large initial size of the corpus, we reach a steady state behavior with respect to popularity of objects requested by each client. Thus, additional requests issued for smaller numbers of clients have negligible impact on the metrics we are interested in.

For each client c , we monitor the following:

- h_c , the number of requests it serves from its own cache (local hits),
- s_c , the number of requests that are served by another client (side-loads),
- m_c , the number of requests that are served by the service (client cache misses),
- b_c , the average bandwidth (over the length of the experiment) used to serve content to other clients.

The metrics examined in this evaluation are the following:

- The average local cache hit ratio: $\sum_c \frac{h_c}{h_c + s_c + m_c} / n$
- The collective (global) hit ratio of the client caches: $(\sum_c h_c + s_c) / (\sum_c h_c + s_c + m_c)$
- The average bandwidth spent per client on serving content: $\sum_c b_c / n$

A local cache hit means that the client can serve an object out of its own cache. A global cache hit means that the server is invoked but the object is served from

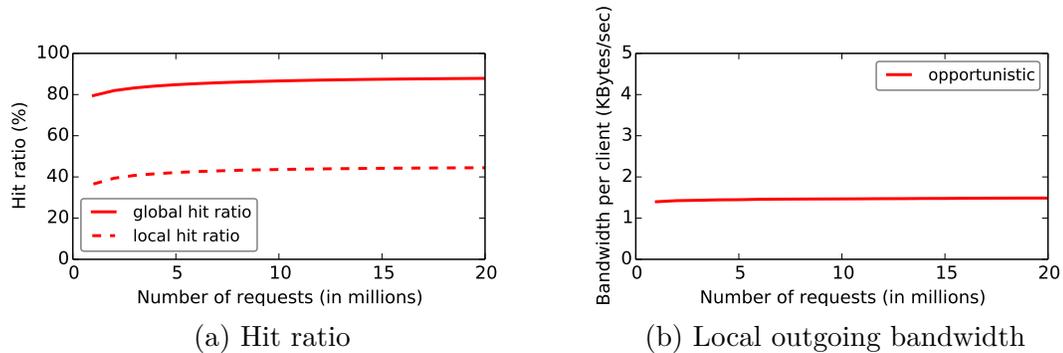


Figure 4.1: Average local hit ratio per client and global hit ratio of the collective cache (a), and average bandwidth per client (b) for the opportunistic approach with 10,000 clients as time passes.

the cache of a client. A miss in both means that the server has to serve the object’s content.

An important consideration in our experiments is bias caused by the initial conditions of the simulation, in particular the initial size of the corpus. The fraction of objects that can fit in the collective cache is inversely proportional to the size of the corpus. In addition, the selection probabilities of each rank in the Zipf distribution change substantially as the corpus size increases for the first few million objects. To reduce the effects of an initially small corpus, we initialize the corpus with 10 million objects.

In Figure 4.1 we observe how the previously described metrics evolve over time for the opportunistic approach with a configuration of 10,000 clients. The corpus is initialized as previously mentioned and the client caches start empty. The x-axis shows the number of requests issued to the service from the beginning of the experiment. A simulated second contains $\sim 2,000$ requests. As the figures show, each metric stabilizes after approximately 10 million requests. This stabilization behavior is similar for all cache placement algorithms discussed in the chapter.

In the following simulations, we begin each run with a 10 million request warm up phase (~ 75 simulated minutes), and then collect measurements for another 10 million requests.

We now summarize the various parameters that affect our simulations. We categorize them in three groups: First, there are those related to the cache placement algorithms such as the minimalistic*'s popularity threshold and the replication factor of the basic proactive approach. Both determine how many copies of an object should be cached in the collective cache, but they target different sets of client caches.

Second, there are the parameters associated with the *workload generation*. These parameters are: i) The number of clients, or size of the client relationship graph, which ultimately determines the size of the system and collective cache. ii) The neighborhood access probability NAP , set to 0.8, which determines the user requesting a particular key. iii) The percentages of read and write requests in the workload set to 95% read and 5% write requests for all experiments. iv) The skew of Zipfian distribution that selects the next key to be requested, fixed at 1.1. v) The key and value sizes (in bytes) for each object that are determined by the Generalized Extreme Value and Generalized Pareto distributions respectively. The former has parameters $\mu = 30.7984$, $\sigma = 8.20449$, and $k = 0.078688$ and the latter has $\theta = 0$, $\sigma = 214.476$, and $k = 0.348238$. Both distributions are taken from [10]. vi) The nearest neighbor model's parameters, that is, the probability $u = 0.96$, and the number of pairs $k = 1$, used for generating client relationship graphs that mimic real social networks.

Finally, there are the parameters that are specific to the simulations we ran. These concern the cache size of each client, which is set to 5MB for all clients, and

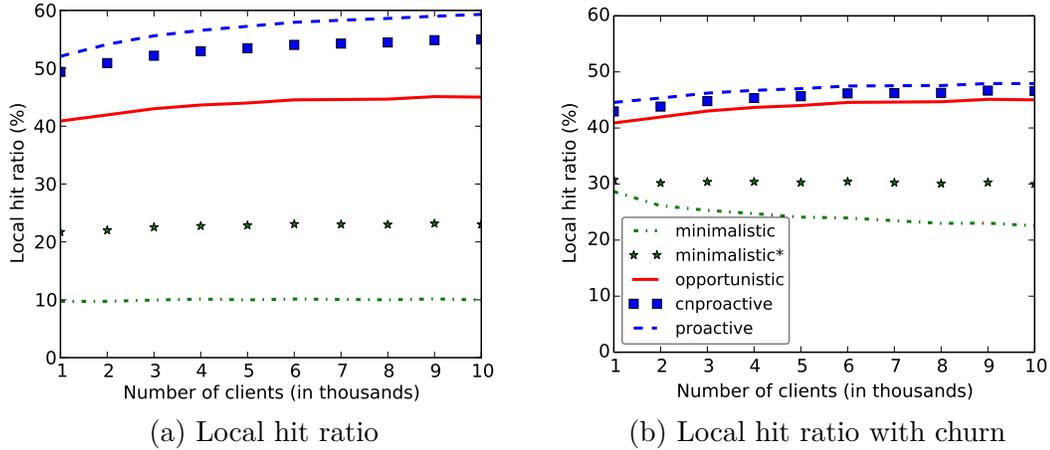


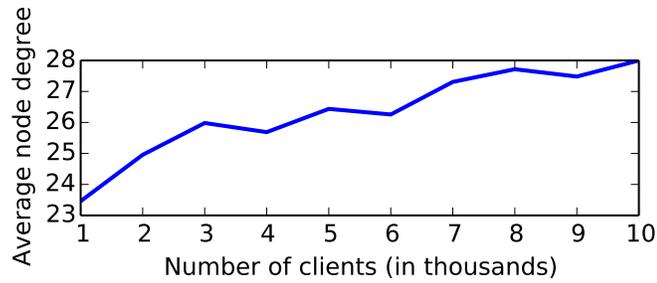
Figure 4.2: Average local hit ratio per client (a) and local hit ratio running under a churn rate of 0.1 (b) for a varying number of clients.

the session window. The latter is expressed in number of requests received by the service before a client who has not issued any requests is considered as offline. The session is used to tune the churn in the experiments presented later in the section.

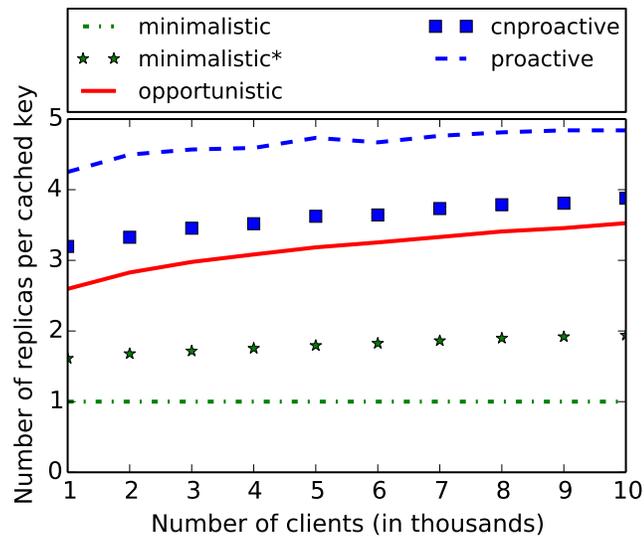
4.2.2 Base case comparison

In this section, we compare the strategies presented in Section 3.3. In the simulations, clients of the service remain online throughout the experiment once they have opened a session (that is, no churn). Clients start issuing requests at different times however. Nevertheless, if there is a client caching the requested object, then a side-load will be successful.

Some of the approaches described in Section 3.3 are parameterized. Minimalistic* uses a threshold parameter that determines whether it creates a new copy or simply moves the requested object from the caching client to the requesting one. For all configurations discussed in our evaluation, we set the threshold to 0.1.



(a) Average node degree of synthetically generated graphs.



(b) Average number of replicas per cached key.

Figure 4.3: Average node degree (a) and average number of replicas per key (b) for a varying number of clients.

The proactive approach is parameterized with a replication factor that determines the fraction of the neighborhood of the owner of the requested object to which the object will be pushed. In the following experiments, all runs of the proactive algorithm have a replication factor of 1, that is, the requested object is pushed to all neighbors excluding clients that already cache the object or are offline. See Section 4.2.4 for more information on how the replication factor affects the proactive approach performance and cost. Our proposed proactive variant

“common neighbors proactive” is denoted as “cnproactive” on all figures that follow.

Local hit ratio

In Figure 4.2a, we show how the average local hit ratio varies with an increasing number of clients. We see that as the number of clients increases, local hit ratio increases slightly for all but the minimalistic algorithm. This is a consequence of the workload model that we use and of the way that the client relationship graph grows as the number of clients increases. Each object is likely to be requested by the neighborhood of its owner because the NAP probability is 0.8 in our experiments. As the number of clients increases, the neighborhood size grows as depicted in Figure 4.3a. In Figure 4.3b, we observe that an increase in the number of clients results in a higher number of object replicas per key.

Since the popularity of each object in each configuration remains the same, but the number of users caching it increases, there is a greater number of users that are likely to find it in their cache, in turn resulting in an increased local hit ratio. Note that this slight increase is not observed for the minimalistic approach because it only creates a single cached copy per object regardless of its popularity or of the number of clients requesting the object.

The proactive algorithm performs best because it creates copies of the accessed object on clients that are more likely to access them. By targeting the right clients, the proactive algorithm increases substantially the likelihood that a future request for an object will be found in the requesting client’s cache. With respect to the local hit ratio, the remaining approaches are ordered according to the number of replicas they create per key (Figure 4.3b). The more replicas for an object, the more

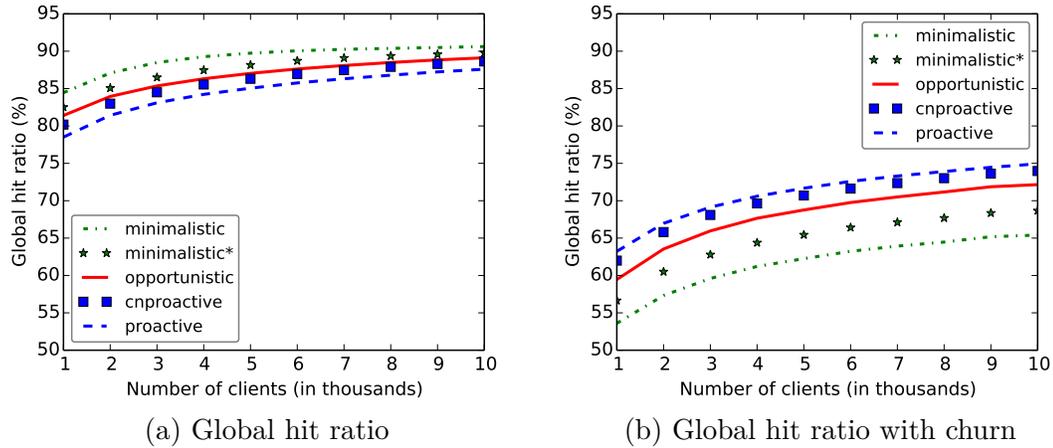


Figure 4.4: Global hit ratio (a) and global hit ratio running under a churn rate of 0.1 (b) as the number of clients increases.

clients caching it and thus the higher the local hit ratio. The proactive approaches perform considerably better, with more than 10% improvement over the reactive techniques (the opportunistic, minimalistic, and minimalistic* algorithms).

Global hit ratio

The global hit ratio is the hit ratio of the collective cache and heavily influences load on the service. Figure 4.4a shows global hit ratio as the number of clients increases. More clients means a larger collective cache and thus a higher global hit ratio. The hit ratio does not reach 100% because there is a steady stream of new objects that are being introduced. Although the proactive and common neighbors proactive approaches perform worse than the other techniques, their respective hit ratio is only $\sim 4\%$ lower than the minimalistic algorithm.

The minimalistic algorithm exploits the collective capacity of the caches best because it creates no duplicates. As we will see in Section 4.2.5, the performance of the minimalistic approach with respect to this metric is very sensitive to churn

as only one copy of an object is maintained. The other algorithms create multiple copies for popular objects and thus have less cache capacity for storing additional objects. This is shown clearly in Figure 4.5, which depicts the ratio of objects cached at the end of the simulation over the total number of objects cached during simulation. The larger the number of replicas that an algorithm creates, the fewer objects that can be stored in the collective cache.

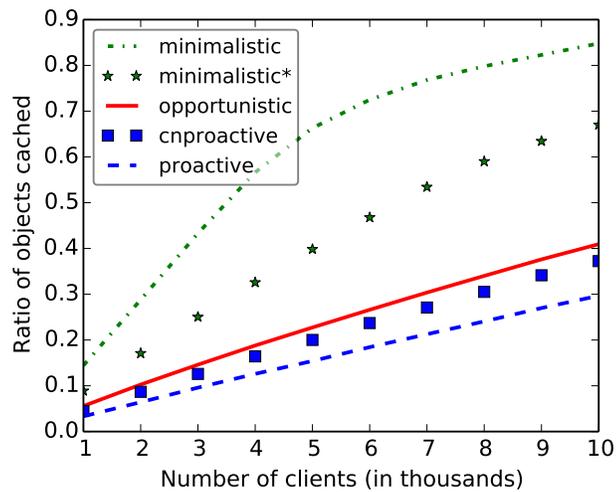


Figure 4.5: Ratio of objects cached at the end of each simulation over the full set of objects requested for a varying number of clients.

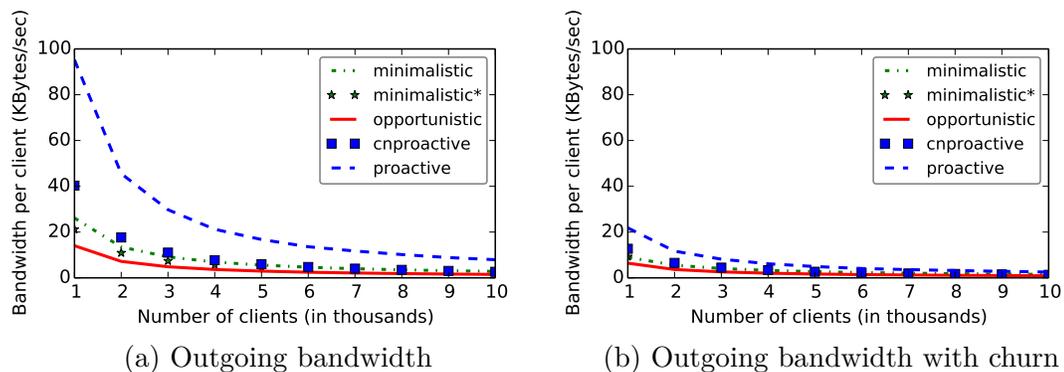


Figure 4.6: Average outgoing bandwidth per client (a) and outgoing bandwidth running under a churn rate of 0.1 (b) for a varying number of clients.

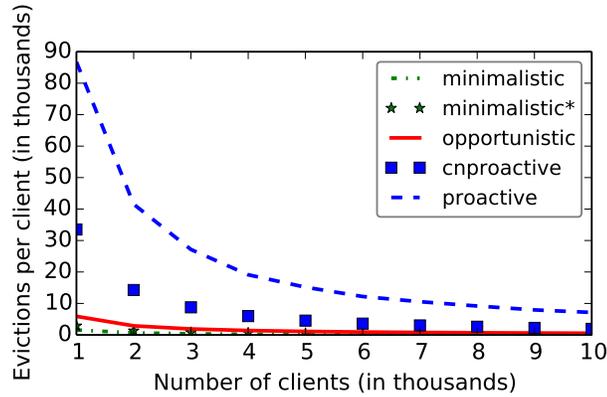


Figure 4.7: Average number of evictions per client for a varying number of clients.

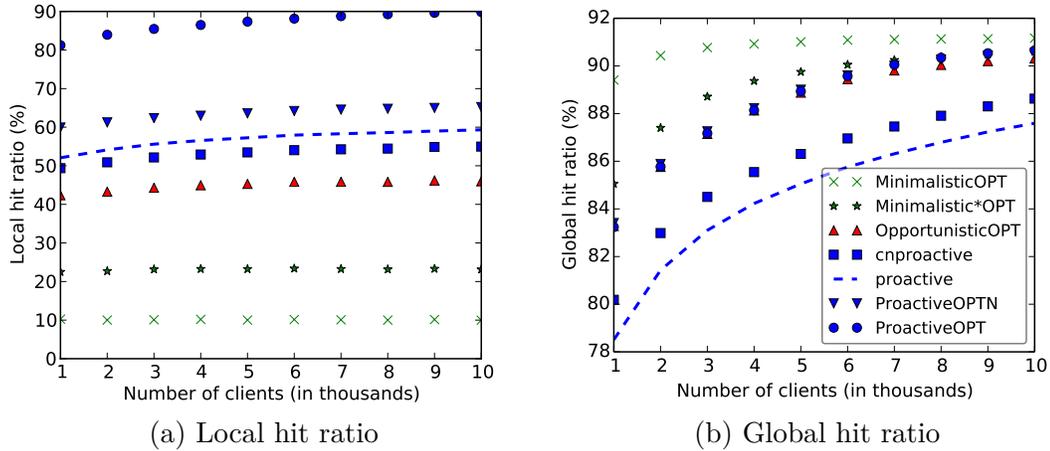


Figure 4.8: Local hit ratio per client (a) and global hit ratio of the collective cache (b) of optimal algorithms as the number of clients increases.

Client bandwidth

Figure 4.6a shows average outgoing bandwidth per client as a function of the number of clients. Local outgoing bandwidth consists of two components: side-loading and pushing. The client bandwidth cost is proportional to the sum of those components. Naturally, a higher local hit ratio lowers frequency of side-loads and pushing (objects are only pushed to caches that do not contain the object). With that in mind it is easier to see why the opportunistic approach performs best. It performs no pushing while it keeps the number of side-loads low by creating

a number of cache replicas that increases with demand. The minimalistic and minimalistic* approaches cannot perform as well with regards to client bandwidth, because they create fewer replicas and thus incur more side-loads.

The proactive approaches push content, which, as can be observed from Figure 4.6a, is more expensive than just side-loading. The reason for that is twofold: First, for each side-load, the proactive algorithms will additionally spend bandwidth pushing to a number of clients (the proactive approach more than the common neighbors variation). Although this results in increased local hit ratio, which saves bandwidth, it also results in a higher eviction rate.¹ This can be seen in Figure 4.7, which shows that the proactive approaches aggressively evict objects from the caches. Second, more evictions will result in more side-loads and thus more pushes as well. Consequently, the proactive approaches spend considerable bandwidth refilling the caches.

The common neighbors proactive approach has comparable performance as the opportunistic one. It achieves this by limiting the number of replicas it creates per side-load to a typically small fraction of the neighborhood (common neighbors only). At the same time it maintains a substantially higher local hit ratio per client, which saves additional bandwidth from future requests served locally on the requesting client.

With all caching approaches, the average bandwidth for side-loading and pushing decreases as the number of clients grows. This is because more clients imply a larger capacity of the collective cache. As the fraction of the corpus that can be

¹Here we refer only to evictions that happen when a client's cache is full and a new object needs to be cached. Evictions initiated by the service are not considered. Unless stated otherwise, evictions refer to those that happen due to the cache replacement policy.

stored in the cache increases, the number of evictions decreases (Figure 4.5). As a result, less side-loading as well as less pushing is needed.

4.2.3 Optimal case comparisons

In this section, we consider results of the optimal versions of the previously examined algorithms. This set of experiments identifies the best performance possible on the metrics discussed. Figure 4.8 depicts the results of these simulations. The optimal version of each algorithm is denoted by the name of the algorithm followed by the “OPT” suffix. Note that for the reactive approaches (opportunistic, minimalistic, and minimalistic*) their optimal counterparts are only optimal with respect to their eviction policy. The proactive approaches are also optimal with respect to their pushing strategies—they push content to exactly those clients that will be requesting this content in the future. We denote by “ProactiveOPTN” the algorithm that performs optimal pushing for each object within the neighborhood of the owner. We include the simple proactive and common neighbors proactive approaches for comparison purposes.

Figures 4.8 and 4.9 expose the potential of the proactive approaches for high local hit ratios, global hit ratios, and low client bandwidth overheads. While the reactive approaches are only marginally improved by an optimal eviction policy, the proactive schemes get a substantial boost on all metrics by pushing content optimally.

The optimal proactive approach (ProactiveOPT) achieves 90% local hit ratio for 10,000 clients and perfect workload knowledge. If, however, we contain the optimal pushing for each object within the neighborhood of its owner (Proac-

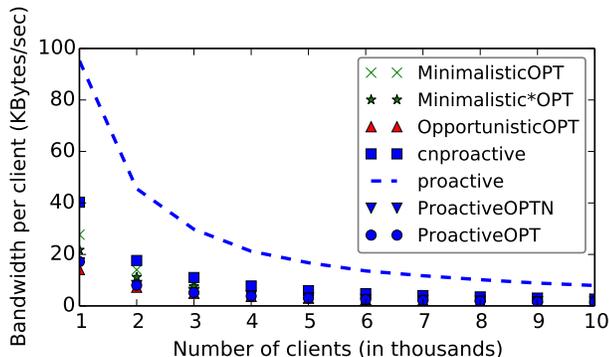


Figure 4.9: Average outgoing bandwidth per client for optimal algorithms as the number of clients increases.

Strategy	LHR	LHR-OPT	GHR	GHR-OPT	B/W	B/W-OPT
Minimalistic	9.98	9.97	90.62	91.17	2.81	2.83
Minimalistic*	23.03	23.14	89.79	90.60	2.31	2.33
Opportunistic	45.02	45.97	89.11	90.31	1.51	1.51
Common Neighbors	54.97	65.21	88.62	90.54	2.64	1.49
Proactive	59.31	89.86	87.59	90.65	7.92	1.46

Table 4.1: Local hit ratio (LHR), global hit ratio (GHR) and average client bandwidth in KB/s (B/W) for each strategy without churn and optimal variants for 10,000 clients. The optimal variants of common neighbors and basic proactive approaches correspond to ProactiveOPTN and ProactiveOPT (from Figures 4.8a, 4.8b, and 4.9) respectively.

tiveOPTN), then the situation is drastically different. In fact, our non-optimal proactive approach and common neighbors proactive approaches approximate the ProactiveOPTN approach relatively well. The differences are within 7 – 10%. This is promising because confining ourselves to the neighborhood of the content owner is a reasonable choice given our target workload. Additionally, attempting to predict requests by clients outside of the owner’s neighborhood may be harder to achieve reliably and is less scalable.

4.2.4 Replication factor

We now investigate the effect of the replication factor (proportion of the neighborhood that caches an object) on the proactive approach. We run a set of simulations with 10,000 clients and vary the replication factor between 0.1 and 0.9. The results of this experiment can be seen in Figure 4.10. We omit a replication factor of 0, because in this case the algorithm becomes the opportunistic approach (no pushing takes place); the results for a replication factor of 1 have been presented in Figures 4.2a, 4.4a and 4.6a.

The findings show that as the replication factor increases both local hit ratio and average client bandwidth increase. This is because the number of replicas created increases with the replication factor (more pushing occurs) and thus more clients are likely to find the content of their neighbors in their caches (Figure 4.10a), thereby increasing the local hit ratio. The additional client bandwidth required by a higher replication factor is fairly small. The global hit ratio drops with the increase of the replication factor because the increasing number of object replicas take more space in the collective cache, leaving less “room” for additional objects to be cached. The decrease in global hit ratio is on the order of 1%, small compared to the corresponding increase in local hit ratio ($\sim 13\%$).

4.2.5 Churn

In Figures 4.2, 4.4 and 4.6, we present the results from a set of experiments where there is non-negligible churn in the system. We model churn as the fraction of clients leaving the system within a period of time. A client is considered to have left the system if the last time it has issued a request was before a predetermined session

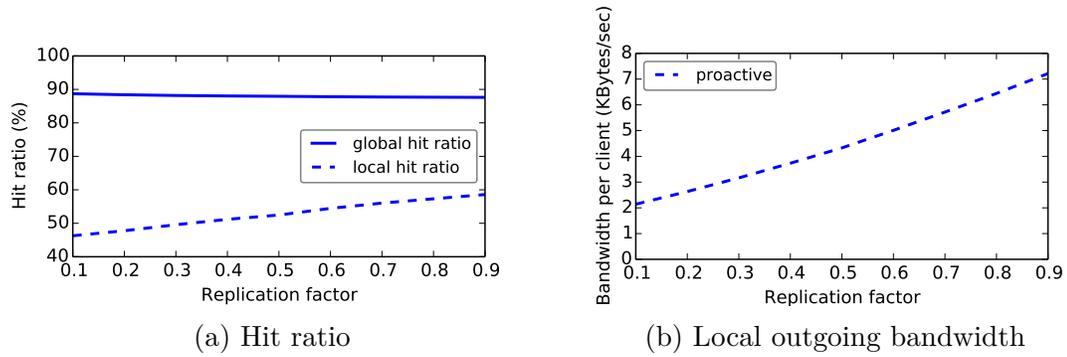
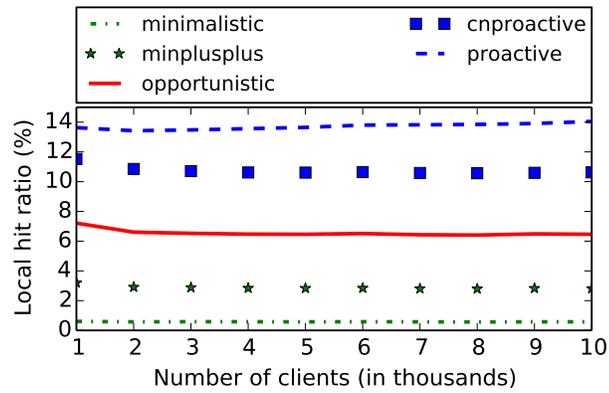


Figure 4.10: Average local hit ratio per client and global hit ratio of the collective cache (a) and average bandwidth per client (b) for the proactive approach running with 10,000 clients as the replication factor increases.

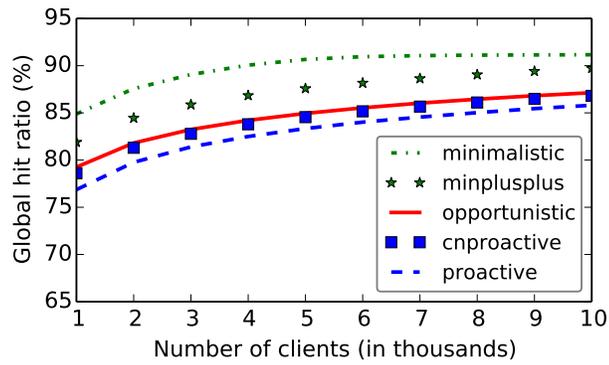
timeout period. We simulate a variety of churn rates by assigning appropriate values to the session timeout period.

In Figure 4.4b we depict the results of the simulations for a churn rate of 0.1. The global hit ratio in Figure 4.4b has decreased substantially for all algorithms compared to the results without churn shown in the same figure (Figure 4.4). This is expected because increased churn means increased chance that a client goes offline soon after it caches an object. This in turn means that a subsequent request for the object will more likely be served by the service. As mentioned in Section 4.2.2, the minimalistic approach is the most sensitive to churn due to the low number of replicas it generates. The more replicas created by an approach, the less vulnerable it is to churn with respect to the global hit ratio. This is why the proactive approach outperforms the other schemes. Consequently, the global hit ratio of the algorithms is strongly correlated with the number of replicas per key they create (Figure 4.3b).

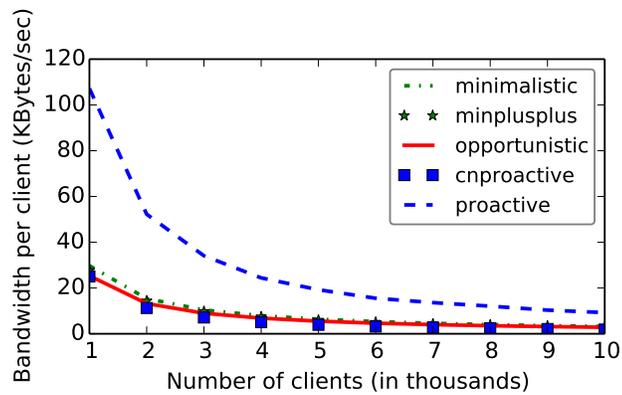
The results regarding clients' outgoing bandwidth in Figure 4.6b are similar to those of the base case (Figure 4.6a). The main difference is in the scale of the bandwidth ranges, which have substantially shrunk. This is because a large number



(a) Local hit ratio



(b) Global hit ratio



(c) Outgoing bandwidth

Figure 4.11: Average local hit ratio (a), global hit ratio (b) and average outgoing bandwidth (c) per client running under $NAP = 0.2$ for a varying number of clients.

of requests are now served by the service and thus clients spend less bandwidth serving content. (Recall that the service bandwidth is not accounted for in the outgoing client bandwidth metric.)

The more surprising results are those regarding the local hit ratio in Figure 4.2b. Observe that both proactive approaches are negatively affected by churn (Figure 4.2a). This is because the rate of clients going offline reduces the number of replicas they can create. This leads to a lower likelihood of a client finding content in its own cache.

The opportunistic approach, on the other hand, is not affected by churn, giving identical results as when clients remain online (Figure 4.2a). The reason is twofold: (i) In our implementation the cache of each client is persistent throughout the multiple sessions the client may open during each simulation run. This is a reasonable assumption given that we target client caches built on their browsers' web storage, which is persistent across sessions. (ii) The opportunistic scheme creates a replica of the requested object on the requesting client whenever a local cache miss occurs. The requesting client is always considered online because it opens or continues a session by virtue of issuing a request. Consequently, churn does not affect the number of replicas it can create and the local hit ratio remains the same.

The minimalistic and minimalistic* approaches benefit from churn with respect to the local hit ratio metric. If a client caching an object goes offline, the next request for the object will generate a new cache copy (from the service's perspective, no client exists that caches the data). When the client that caches this object comes online and starts a new session, two copies will be available in the collective cache because caches carry over different sessions in the same simulation run. Con-

Strategy	LHR (Churn)	GHR (Churn)	B/W (Churn)
Minimalistic	22.54	65.40	1.51
Minimalistic*	30.00	68.66	1.35
Opportunistic	45.02	72.15	0.92
Common Neighbors	46.59	73.98	1.33
Proactive	47.90	74.93	2.53

Table 4.2: Local hit ratio (LHR), global hit ratio (GHR) and average client bandwidth in KB/s (B/W) for each strategy with moderate churn for 10,000 clients.

sequently, the minimalistic approaches can generate more replicas per key under churn, which results in an increase of local hit ratio.

We observed similar results for higher levels of churn rate, though the differences were more pronounced. Minimalistic and minimalistic* did even worse with respect to global and local hit ratio. In addition, the gap between proactive and opportunistic approaches increased for the global hit ratio and decreased for the local hit ratio. The results regarding bandwidth were similar to those in Figure 4.6b with decreased bandwidth spent in pushing and side-loading.

4.2.6 Neighborhood Access Probability

So far we have investigated the behavior of client caches under workload characteristics that substantially skew object accesses within the neighborhood of the owner of the object. The magnitude of this skew is determined by NAP , which for the results presented so far was pinned to 0.8 as literature [16] suggests for OSNs. In this section, we observe the performance characteristics of the previous caching strategies for $NAP = 0.2$, that is, a small workload skew. This means that 80% of requests for an object come from randomly selected clients outside the object owner’s neighborhood in the client relationship graph. Note that given the graph sizes and the average neighborhood sizes shown in Figure 4.3a, setting NAP to

such a low value renders the requesting client being chosen almost uniformly at random, 80% of the time. The objects' popularity and sizes distributions remain the same and the read/write ratio is preserved.

We perform the same experiments as in Section 4.2.2. In Figure 4.11a, we observe how local hit ratio is affected by the size of the system. Local hit ratio has drastically decreased for all algorithms compared to Figure 4.2a, because objects are now accessed by a wider set of clients. The ordering of the protocols with respect to their local hit ratio is still as in Figure 4.2a because strategies that create more replicas have a better local hit ratio. For all algorithms the local hit ratio is not much affected by the number of clients.

Figure 4.11b shows results for the global hit ratio. The results are similar to our prior experiments (Figure 4.4a), the main difference being that all approaches show slight decrease in global hit ratio. This is because object accesses are more spread over the set of clients, resulting in more replicas per object and less cache capacity for storing additional objects.

The average outgoing bandwidth shown in Figure 4.11c is also similar to previous experiments (Figure 4.6a). Two points are worth noting: First, all approaches spend more bandwidth because access is more spread out and more side-loads as well as pushing occur. Second, the common neighbors proactive approach performs similar to the opportunistic approach because it is less likely that there are common neighbors between the requesting client and the owner of the requested object for small *NAP*.

Strategy	# replicas / object	# evictions / client	% objects cached
Minimalistic	1.0	29	84.80
Minimalistic*	1.94	132	67.01
Opportunistic	3.52	536	40.95
Common Neighbors	3.87	1926	37.23
Proactive	4.83	7140	29.67

Table 4.3: Average number of replicas per objects, evictions per client, and ratio of objects cached for each caching strategy for 10,000 clients.

4.2.7 Results Summary

Our simulations demonstrate how we can employ workload-specific information to improve performance and efficiency of cooperative caching. With small overhead on service load ($\sim 2\%$) and cost in client bandwidth, we can substantially improve client perceived latency (Figures 4.2a, 4.4a, and 4.6a). The relative costs of proactive replica creation and placement decrease as the system size increases. This suggests that our approaches are promising for systems of larger scale. By controlling the amount of proactivity in our strategies (Figures 4.10a and 4.10b), we can drive these costs to be competitive to the widely used opportunistic approach while retaining most of the benefits of the proactive approach. In addition, our proactive approaches approximate the optimal cache placement strategy on local hit ratio within a relatively small range ($\sim 7\%$) when the optimal placement is confined within the neighborhood of the owner of the requested object (Figure 4.8a). Our proactive approaches handle churn more gracefully (Figures 4.2b, 4.4b and 4.6b) because of the increased number of replicas created per object (Figure 4.3b). Finally, our approaches remain competitive even under workloads with lower locality (Figure 4.11). Tables 4.1, 4.2, and 4.3 summarize the previous results for 10,000 clients and $NAP = 0.8$.

4.3 Concluding remarks

In this work we studied the cache placement problem for a cooperative cache built on the clients of an Online Social Network. We considered a service that maintains a directory of content locations and that has access to the social links between clients. The service provides the clients with hints about where content can be found as well as where it should be cached. Given this system model, we proposed proactive cache placement strategies that leverage social links between clients.

We evaluated the proposed proactive schemes using simulations on synthetically generated graphs and workloads whose characteristics match those of real OSNs and compared them with various baseline approaches. Our findings show that proactively caching content where it is most likely to be requested can achieve substantial improvements in hit ratio over commonly used approaches at moderate overheads.

We also implemented optimal versions of the various cache placement strategies we considered. These approaches know the entire workload ahead of time. In one of our proactive variants, the hit ratio is only $\sim 7\%$ lower than optimal.

Finally, we explored the effects of churn and showed that our proposed proactive approaches are effective even under moderate churn.

CHAPTER 5
**TURTLE CONSENSUS: A CASE OF REACTIVE PROTOCOL
SWITCHING**

Consensus is a basic building block in middleware configuration services [20, 53]. While such services are designed to tolerate crash failures in asynchronous settings, they may not stand up well to Denial-of-Service (DoS) attacks. Specifically, malicious clients can carefully craft workloads that substantially degrade the performance of many state-of-the-art consensus protocols. By exploiting protocol-specific vulnerabilities, attackers can constantly force the protocol participants to slow execution paths [31]. In this paper, we investigate designing consensus protocols that provide acceptable performance under DoS attacks that aim to saturate the bandwidth of protocol participants.

We propose a new asynchronous consensus protocol that we call *Turtle Consensus*. Turtle Consensus employs previously proposed crash-tolerant consensus protocols and exploits their diverse characteristics by switching between protocols from round to round. Some protocols are fast under benign conditions but their performance suffers greatly under attack. Other protocols may not be as fast under benign conditions, but their performance may actually benefit from naive attacks. By reconfiguring the consensus protocol on-the-fly we can achieve the best of both worlds: excellent performance in benign scenarios and acceptable performance while under attack, even if the client workload is high. We evaluate Turtle Consensus against adversarial scenarios where at most one process may fail and show that we can achieve better performance than existing crash-tolerant protocols under attack.

5.1 Introduction

Consensus is one of the most critical components for many fault-tolerant services. The problem of agreement on state by a set of distributed processes is at the core of various replication protocols like state machine replication [63, 96]. As a result, consensus protocols have been heavily studied from the early beginnings of distributed systems and still receive considerable attention [77, 81]. In practical settings where communication is asynchronous and faults of various severity levels are a reality, previous work has focused on crash-tolerant protocols [24, 77] and BFT protocols [15, 22, 33, 61] that can handle a limited number of failures. One problem with these protocols is that their design favors primarily the “common” or *gracious* executions where no failures arise while only guaranteeing weak timeliness guarantees such as eventual progress or liveness in the faulty cases. Indeed, [31] shows that state-of-the-art protocols become almost unavailable under certain *denial-of-service* (DoS) attacks by clients, the replicas of the replicated service, or both. Certain design decisions made for good performance under gracious executions turn out to be frail and result in bad performance while under attack.

One way around the problem is to build a front-end tier of stateless servers that relay incoming requests to the fault-tolerant service, checking the validity of incoming requests and filtering out the invalid ones. This is a solution that is not only expensive, but does not really solve the problem as the attacker may be a compromised front-end server.

The work in [31] proposes a new Byzantine consensus protocol, which addresses various of the issues exposed. In this work, motivated by similar concerns, we pose the question of how can we can leverage the design of existing asynchronous crash-

tolerant consensus protocols in order to provide acceptable performance while under attack.

Diversity has been proposed as a mean for building attack-tolerant systems [91]. In this proposed framework, different replicas may run different code or variations of the same protocol (diversity in space) to mitigate correlated failures and thus limit the damage from exploiting existing vulnerabilities. In addition, replicas may change their protocols on-the-fly (diversity in time) as attacks or problematic workloads are detected. In this spirit, we propose *Turtle Consensus*, a *moving target defense* scheme for attack-tolerant consensus in asynchronous environments.

We present Turtle Consensus, a round-based consensus protocol that embraces *diversity-in-time* by dynamically changing the protocol at each round. For this, Turtle Consensus leverages existing consensus protocol designs. Many fault-tolerant consensus protocols have been proposed, each optimized for different conditions or metrics. Approaches vary in many ways. For example, some consider scale, ranging from datacenter-scale systems [24] to wide area deployments [72, 77]. The resulting protocol techniques range from leader-oriented [64, 26] to fully decentralized and randomized [15] protocols. As a result, Turtle Consensus can be used in a variety of settings by limiting the selection of protocols in each round to those applicable to the setting and metric of interest. Here we are focused on achieving acceptable performance under DoS attacks that aim at exhausting the bandwidth of participants of the protocol. We briefly discuss attacks that exhaust CPU resources of participants in Section 5.5. The idea of alternating protocols has been previously explored [4, 106, 68, 47, 32] though was mainly focused on high performance under variable contention. These constructions do not address DoS.

The chapter is organized as follows: In Section 5.2 we describe the system model and provide some background on consensus protocols. In Section 5.3 we present our proposed Turtle Consensus protocol and demonstrate its correctness. Then in Section 5.4 we give some implementation details of Turtle Consensus and describe how we implemented a state machine replication protocol on top of it. We describe our experimental results in Section 5.5. Section 2.2 briefly discusses related work. Finally, we conclude in Section 5.6.

5.2 Background

In this section we describe the system model, we specify the attacker’s capabilities, and provide background of the protocols we are using underlying Turtle Consensus.

5.2.1 System Model

Our distributed system is modeled as a set of *processes*, $N = p, q, \dots$. Each process can be viewed as a state machine with a potentially unbounded set of states, making possibly non-deterministic transitions according to some protocol. The protocol specifies the communication between processes, which happens via message passing. A process can be either correct or faulty. A correct process faithfully follows the protocol and is guaranteed to make progress given that the conditions specified by the protocol at any given step are eventually met. Faulty processes may crash (stop executing the protocol) at any time. We assume that up to the point of its crash the faulty process faithfully executes the protocol. In this work, we do not concern ourselves with the reintegration of a failed process into

the system. We impose an upper bound $t < |N|$ on the number of crash failures that may occur during operation of the system.

Communication between correct processes is reliable, meaning that a message sent by a correct process to another correct process will eventually be delivered. We also assume that there exist time bounds on communication delays and the time that processes take to execute transitions. However, we do not assume that these bounds are known, and they can be arbitrarily large. Note that this is a stronger assumption than asynchrony (and the well-known impossibility result [42] does not apply), but much weaker than assuming known bounds on latencies. We shall refer to it as *weak synchrony*.

The adversary has total control over the timing and identity of the processes that can crash, though t limits how many crashes it can cause. In addition, the adversary can delay communication and deliver messages in any order, but it must yield to the reliable network constraint stated in the previous paragraph.

5.2.2 Denial-of-Service

The adversary can attack the system by issuing *Denial-of-Service* (DoS) attacks. DoS attacks saturate the bandwidth resources of one or more processes, degrading their ability to participate in the protocol. In this way, the adversary can introduce communication delays or control how various processes progress through their transitions. By orchestrating the participation (or more accurately lack thereof) of a limited number of processes in the system, the adversary aims to drive the protocol execution into the most computationally expensive paths.

There are other types of DoS attacks. For example, an attacker could use well-formed requests that are known to impose substantial load on one or more of the processes running the consensus protocol. CPU attacks using legitimate requests can be mitigated to a certain degree using client puzzles [11].

5.2.3 Consensus Protocols

A consensus protocol, CP , is a protocol run by the processes in N , each of which has an input value from some globally known set, V , and whose goal is for all correct processes to agree on one of the input values [41]. Every process in N can *propose* or *decide* a value and a decision is irrevocable. A consensus protocol satisfies the following properties:

- **Agreement:** If any two processes decide, they decide the same value.
- **Validity:** If a process decides a value $v \in V$, then v must have been the input value of some process.
- **Termination:** All correct processes eventually decide.

Different consensus protocols can vary significantly in their details regarding the minimum number of processes required, the state each process maintains, and how it manages that state. Our approach relies on the processes switching between protocols so we abstract some of these details away to facilitate our description.

In this work, we only consider consensus protocols that operate in rounds. To the extent of our knowledge most consensus protocols in the literature operate this way, for example [64, 22, 26, 15]. The inputs of the processes comprise the input

of the round. We model the output of a process running a round of CP as a tuple $(s, v) \in \{\mathbf{D}, \mathbf{M}, \perp\} \times V \cup \{\perp\}$ where $\mathbf{D}, \mathbf{M}, \perp \notin V$ are special values. Let u_p be the output of process p after completing a round of CP :

- If $u_p = (\mathbf{D}, v)$ where $v \in V$, then p has decided v .
- If $u_p = (\mathbf{M}, v)$ where $v \in V$, then if some process has decided, it must have decided v . Formally: $\exists p' \in N, v' \in V : u_{p'} = (\mathbf{D}, v') \Rightarrow v' = v$.
- $u_p = (\perp, \perp)$ then no process can decide. Formally: $\nexists p' \in N, v \in V : u_{p'} = (\mathbf{D}, v)$.

Notice that (\mathbf{D}, v) denotes a decided process, while (\perp, \perp) implies no decision could be made in the past rounds. (\mathbf{M}, v) intuitively means that the respective process has not decided yet but it knows that if a decision has been made in the round, then it must be for value v .

We now describe some key invariants on the outputs of quorums of processes running a round of protocol CP . Before doing so, recall that a quorum in a consensus protocol is informally a subset of processes that is sufficient for a decision to be made. A *quorum system* $\mathcal{Q} \subseteq 2^N$ is a non-empty set of quorums, every pair of which has non-empty intersection. Note that different consensus protocols may employ different quorum systems for ensuring the previously mentioned properties.

We define a quorum system \mathcal{Q}_{CP} with respect to a consensus protocol CP as a quorum system on the processes running CP such that a process cannot make a decision before receiving messages from the processes in a quorum $Q \in \mathcal{Q}_{CP}$.

A consensus protocol is called *t -crash-resilient* when it is correct (implements its specification) even in the case where at most t processes experience crash failures.

In an asynchronous environment crashed processes cannot be reliably detected by correct processes. For the rest of the paper we will refer to *t-crash-resilient* consensus protocols simply as *t-resilient* protocols. For any *t-resilient* protocol the following property holds:

Property 1. *Let CP be a t -resilient protocol running on a set of processes, N such that $|N| > t$. Any quorum $Q \in \mathcal{Q}_{CP}$ must satisfy $|Q| > t$.*

Property 1 stems from the fact that CP is t -resilient. By contradiction, assume a quorum Q of t or fewer processes exists. Assume a configuration in which the processes in Q propose v while the remaining processes propose v' , $v' \neq v$. Because Q is a quorum, processes in Q may decide v without communicating with the processes in $N \setminus Q$. Now assume all processes in Q fail, which is possible because $|Q| \leq t$. Assume the remaining processes are correct, and thus they are required to decide by the termination property. Because these processes may not be aware that v was proposed, the only value these processes can decide is v' , which breaks the agreement property. Therefore no quorum smaller than $t + 1$ can exist.

If a decision has been made by some process we know the following:

Invariant 1. *If $\exists p \in N, v \in V$ such that $u_p = (D, v)$ after a round of CP , then $\exists Q \in \mathcal{Q}_{CP} : \forall p' \in Q : u_{p'} = (D, v) \vee u_{p'} = (M, v)$.*

Invariant 1 states that once a decision v has been made, there must be a quorum of processes that have either decided v or know that if a value was or will be decided, it is v .

Invariant 2. *There can never $\exists p, p' \in N$ and $v \in V$ such that $u_p = (\perp, \perp)$ and $u_{p'} = (D, v)$.*

The invariant states that no process can be undecided if a decision has been made. This invariant holds trivially by the definition of (\perp, \perp) .

5.3 Turtle Consensus

We now present our *Turtle Consensus* protocol. Turtle Consensus is a t -resilient consensus protocol that runs in rounds. Each round the system's processes may run a round from a different underlying consensus protocol, as long as they satisfy the invariants described in Section 5.2.

All processes run the same underlying consensus protocol at any given round. To achieve this, the processes either agree on the sequence of protocols ahead of time, or employ a phase at the beginning of each round during which the processes select the consensus protocol to run and its parameters. While we currently only implemented the former, we describe the latter here for completeness. Depending on the outcome of the selection phase, the processes run a round of the selected protocol. Depending on the outcome of their execution they update their value and decision state and proceed to the next round. We denote by $o_p \leftarrow CP(v_p)$ the execution of a round of CP by process p given input v_p and yielding *outcome* o_p . A process' outcome extends the output of the process running a round of CP with the notion of timeouts—a formal definition is given later in this section. In the remainder of this section we will describe Turtle Consensus in greater detail and provide a sketched proof of correctness.

We assume that all processes have common knowledge of N , of the maximum number of crash failures t , and of a set \mathcal{P} of pre-approved, round-based, t -resilient consensus protocols satisfying the conditions described in Section 5.2. We also

assume that N is large enough to satisfy the liveness conditions of any protocol in \mathcal{P} . Every process i runs Turtle Consensus with as input the identifier of the process (*i.e.*, i) and its proposal, $x_i \in V$. At any point in time each process i maintains a state containing:

- its current round number, r_i , initialized to 0;
- its proposal v_i for round r_i , initialized to x_i ;
- the outcome of the round, o_i , regarding the decision state, initialized to (\perp, \perp) at the beginning of each round; and
- a configuration $c_i \in \mathcal{P}$ whose value describes the protocol that will be run by i during r , and is initialized to NULL at the beginning of each round.

The set of possible values of an outcome is defined as:

$$\mathcal{O} = \left\{ \bigcup_{v \in V} \{(\mathbf{D}, v), (\mathbf{M}, v)\} \right\} \cup \{(\perp, \perp), (\mathbf{T}, \perp), (\mathbf{DN}, \perp)\}$$

Values (\mathbf{D}, v) , (\mathbf{M}, v) and (\perp, \perp) for $v \in V$ are defined as the processes' outputs running a CP round in Section 5.2. When $o_i = (\mathbf{T}, \perp)$ then process i has timed out while executing a round of CP and thus failed to compute a value. Finally, if $o_i = (\mathbf{DN}, \perp)$ then i has not learned the configuration of the round and $c_i = \text{NULL}$. We explain how these may occur in more detail later.

In addition to the previous state, each process i knows a function $\mathcal{F}_i : 2^{V \times \mathbb{N}} \rightarrow V$ which takes a non-empty multiset \mathcal{M} of values from V as input and outputs a value from that multiset. We consider \mathcal{M} as a set of tuples $(v, m(v))$ where $v \in V$ and $m : V \rightarrow \mathbb{N}$ is a multiplicity function that given $v \in V$ outputs the number of v 's

occurrences in the multiset. Given a multiset \mathcal{M} , we denote the multiplicity of element $e \in \mathcal{M}$ by $m_{\mathcal{M}}(e)$. We call \mathcal{F}_i the *value-update function* of process i . The only constraint we impose on \mathcal{F}_i is that its output must be a value of the input multiset. Other than that, \mathcal{F}_i can be arbitrary, for example, employing random coin tosses, using the multiplicities of values, and so on. Value-update functions may differ between processes.

We use timeouts to ensure progress under failures. If a process stops receiving any messages for a substantial amount of time at some round r , it times out. Timeouts are used as an unreliable failure detector on which we rely only for liveness. Recall that we assume a weak synchrony model in which communication latency and process transition times have bounds, albeit unknown. Similar assumptions have been used in the literature (for example, [26, 38]) to circumvent the well-known FLP impossibility result of fault-tolerant asynchronous consensus [42]. Our protocol need not know these bounds and yet can still exploit their existence. To do so, we exponentially increase timeouts each round (by multiplying them by a configurable factor larger than 1), such that correct processes eventually make progress within a round without timing out.

The protocol runs for an unbounded number of rounds and eventually reaches a state in which all correct processes have decided. Processes label each message with their current round number r_i and only act upon messages of their current round r_i . Messages from old rounds, either delayed in the network or sent by slow processes that have not caught up, are discarded. Messages from future rounds, that is, from processes that have advanced further in the protocol's execution, are queued to be processed when the receiver reaches that round.

Each round of Turtle Consensus consists of 3 phases:

Phase 1: The processes agree on a configuration determining the protocol, CP , they will run for this round. At the end of this phase it holds $\forall i \in N : c_i = CP \vee c_i = \text{NULL}$. If $c_i = \text{NULL}$, i sets $o_i = (\text{DN}, \perp)$.

Phase 2: If $c_i \neq \text{NULL}$, process i executes a round of CP using its current proposal, v_i , as input. Upon successful completion: $o_i \leftarrow CP(v_i)$. If i times out during CP , it sets $o_i = (\text{T}, \perp)$.

Phase 3: Each process i broadcasts its outcome. If $c_i \neq \text{NULL}$, process i waits for an outcome from each process in a quorum $Q \in \mathcal{Q}_{CP}$. Otherwise i waits for $|N| - t$ outcomes from different processes. Let R denote the set of received outcomes:

Case 1: If $\exists o \in R$ such that $o = (\text{D}, v)$, then process i updates its proposal $v_i = v$, decides v , sets its outcome $o_i = (\text{D}, v)$ and never changes o_i and v_i again in any future round.

Case 2: If $\exists o \in R$ such that $o = (\text{M}, v)$ and $\forall o' \neq o \in R : (o' = (\text{M}, v) \vee o' = (\text{T}, \perp) \vee o' = (\text{DN}, \perp))$, then process i updates its proposal to v .

Case 3: If $(\perp, \perp) \in R$ or $\exists o, o' \in R$ and $v, v' \in V$ such that $(o = (\text{M}, v) \wedge o' = (\text{M}, v')) \wedge (v \neq v')$, then i updates its proposal according to its value-update function \mathcal{F}_i to a value in $\{v_i\} \cup \{v : (\text{M}, v) \in R\}$.

Case 4: In all other cases (*i.e.*, $\nexists v \in V$ such that $((\text{M}, v) \in R \vee (\text{D}, v) \in R)$), then i keeps its current proposal v_i .

The process then moves on to the next round.

Algorithm 2 presents pseudo-code for Turtle Consensus. The *SelectProtocol* function takes the set of pre-approved protocols and a round number and returns either the protocol that should be run by all processes in that round or NULL if the

protocol selection function fails. Messages in the algorithm are encoded as tuples of the form $\langle sender, data, round \rangle$. The *Broadcast* function sends the input messages to all processes in N . Messages for future rounds are buffered by the protocol and delivered once the respective round is reached. To enable such delayed delivery, we keep track of these messages in variable *delayed_msgs* and provide access to it in all 3 phases through the functions *SelectCP*, *CP* (stored in the configuration c_i), and *receive*. Note that operation *receive(delayed_msgs)* delivers messages from either the network or *delayed_msgs* if it is not empty. In the latter case, delivered messages are removed from *delayed_msgs*.

We now describe the three phases in greater detail. Recall that at the beginning of each round, each process sets its $c_i = \text{NULL}$ and if not yet decided, its outcome to (\perp, \perp) . In the first phase the processes need to agree on a configuration, in particular, which protocol to run. There are various ways to decide on the configuration. In the following description, we briefly present some potential ways of implementing *SelectCP*.

A static scheme could select *CP* deterministically according to the current round number. This way all processes in the same round know the same configuration and can participate in the protocol. A more dynamic scheme could assume a deterministically chosen leader for each round, r , say the process with identifier $r \bmod N$. The leader of the round chooses a configuration and then broadcasts it to all other processes. Any process delivering a configuration broadcast for its current round can then start executing the protocol specified in the configuration.

One problem arising with dynamic configuration selection schemes is that some processes may never learn the protocol selected for a round r when they reach that round. For example, in the case where a leader decides and broadcasts the config-

Algorithm 2 Turtle Consensus

```
1: Input:  $x_i, i$ 
2: Output:  $o_i = (D, v)$  where  $v \in V$  the value decided.
3: State:  $r_i := 0; v_i := x_i; o_i := \perp; c_i = \text{NULL}; RecvMsgs := \emptyset;$ 
4:  $wait\_for := 0; R = \emptyset; future = \emptyset$ 
5: while True do ▷ Run forever
6:    $delayed\_msgs = \{\langle j, k, r_i \rangle \mid \langle j, k, r_i \rangle \in future\}$ 
7:    $future := future \setminus delayed\_msgs$ 
8:   if  $o_i \neq (D, v_i)$  then
9:      $c_i := \text{SelectCP}(\mathcal{P}, r, delayed\_msgs)$  ▷ select CP for  $r$ 
10:    if  $c_i == \text{NULL}$  then ▷ timeout occurred
11:       $wait\_for := |N| - t$ 
12:       $o_i := (DN, \perp)$ 
13:    else
14:       $wait\_for = |Q_{c_i}|$ 
15:       $o_i := c_i(i, v_i, delayed\_msgs)$  ▷ run a round of CP
16:    end if
17:     $Broadcast(\langle i, o_i, r_i \rangle)$ 
18:  else
19:     $wait\_for := |N| - t$ 
20:     $Broadcast(\langle i, o_i, r_i \rangle)$ 
21:  end if
22:  while  $|RecvMsgs| < wait\_for$  do ▷ wait for a message
23:    switch ( $receive(delayed\_msgs)$ )
24:      case  $\langle j, k, r_i \rangle$ :
25:        if  $\nexists (j, o) \in RecvMsgs$  then
26:           $RecvMsgs := RecvMsgs \cup \{(j, k)\}$ 
27:        else
28:           $RecvMsgs := RecvMsgs \setminus \{(j, o)\} \cup \{(j, k)\}$ 
29:        end if
30:      end case
31:      case default:
32:        continue ▷ ignore malformed messages
33:      end case
34:    end switch
35:  end while
36:   $R := \{o : (j, o) \in RecvMsgs\}$  ▷ all received outcomes
37:  if  $\exists (D, v) \in R$  then
38:     $v_i := v; o_i := (D, v_i)$ 
39:  else if  $\exists (\perp, \perp) \in R$  or  $\exists (M, v), (M, v') \in R$  then
40:     $v_i = F_i(\{v_i\} \cup \{v : (M, v) \in R\})$ 
41:  else if  $\exists (M, v) \in R$  and  $(\forall o \in R : o = (M, v))$  or
42:     $o = (DN, \perp)$  or  $o = (T, \perp)$  then
43:     $v_i = v$ 
44:  end if
45:   $RecvMsgs := \emptyset; R := \emptyset; r_i := r_i + 1$ 
46:  if  $o_i \neq (D, v_i)$  then
47:     $o_i = \perp$ 
48:  end if
49: end while
```

uration to the rest of the processes, the leader may fail and some processes may never receive CP . We denote this outcome with value, (DN, \perp) . If $i \in N$ has $o_i = (DN, \perp)$ then $c_i = \text{NULL}$ meaning that process i has not computed this round's configuration, and thus it cannot participate in the remainder of the round. To ensure progress under such failures, we employ timeouts and an outcome propagation phase (Phase 3).

In Phase 2, the processes that have successfully computed the common configuration $c_i = CP$ run a round of CP . Note that we can add CP to all messages sent, thus informing receivers of this round's configuration in case they had not learned it in Phase 1. To keep our protocol description simple, we omit such optimizations.

If a process completes a round of CP , it updates its outcome to the output of CP , that is (D, v) , (M, v) for some $v \in V$, or (\perp, \perp) . If, instead, the process times out while executing CP it sets its outcome to (T, v) . Once the outcome of a process for the current round has been computed, it begins Phase 3.

In Phase 3, each correct process i broadcast its outcome and, if $c_i \neq \text{NULL}$, waits for an outcome from each process in a quorum, $Q \in \mathcal{Q}_{CP}$. Otherwise, not knowing \mathcal{Q}_{CP} , i waits for $|N| - t$ outcomes. This is because any set of $|N| - t$ processes is guaranteed to contain a quorum. (If not, the termination requirement of a t -tolerant consensus protocol could be violated.) Once process i has received such a multiset of outcomes, R , there are four cases to consider:

1. If i received a decision outcome, (D, v) , then there is a process that has already decided and thus i adopts the proposal and decides v . In this case, i stops updating its outcome and proposal and keeps those for future rounds it might participate in.

2. If there is an outcome (\mathbf{M}, v') for $v' \in V$ and all other outcomes from processes that executed CP and have not timed out are also (\mathbf{M}, v') , then it is known that if a value has been or will be decided, that value is v' . Thus, i adopts v' as its proposal.
3. If some process declares that no value can be decided in the current round, that is $(\perp, \perp) \in R$, or if there are conflicting outcomes regarding the value expected to be decided next, that is $\exists v, v' : (\mathbf{M}, v) \in R \wedge (\mathbf{M}, v') \in R \wedge v \neq v'$, then no process can have decided in this round or a prior one. Process i then sets its proposal to one of the values seen in R . To choose a value, i uses its value-update function \mathcal{F}_i with input the multiset, \mathcal{M} , of values in R or more formally: $\mathcal{M} = \{(v, m_R((\mathbf{M}, v))) : (\mathbf{M}, v) \in R\}$. If \mathcal{M} is empty, process i keeps its proposal v_i .
4. The only case left is when R does not contain any values, that is $\forall o \in R : o = (\mathbf{T}, \perp) \vee o = (\mathbf{DN}, \perp) \vee o = (\perp, \perp)$. In this case, it is not possible that a decision has been made and thus i keeps its proposal, v_i .

In Phase 3, a process with (\mathbf{DN}, \perp) outcome waits for $|N| - t$ outcomes. Phase 3 ensures that processes that have timed out will be updated on the progress made by the processes that did run CP .

Correctness (Sketch)

Turtle Consensus needs to satisfy the agreement, validity and termination properties described in Section 5.2. To demonstrate these properties we assume that the internal consensus protocols we use in Turtle Consensus are correct and already satisfy the previously mentioned properties.

Turtle Consensus introduces two additional outcomes to existing protocols namely (DN, \perp) , (T, \perp) , both of which typically result due to a timeout occurring during Phase 1 or Phase 2. Timeouts may lead to absence of responses from a quorum of participating processes in a protocol round. Notice that any t -resilient consensus protocol running with fewer processes than the size of any of its quorums satisfies all properties (agreement and validity) but not termination. To satisfy termination in Turtle Consensus, we first need to prove that:

Lemma 5.3.1. *For any unbounded execution of Turtle Consensus, there will be arbitrarily many rounds where at least a quorum of correct processes participate without timing out.*

Proof. First, observe that all processes will reach Phase 3, either by timing out or receiving the messages required to finish the first two phases. In Phase 3 of the protocol, at least $|N| - t$ processes send their outcome to each other because at most t processes fail. Moreover, any process, timed-out or not, needs at most $|N| - t$ outcomes to complete Phase 3 and proceed to the next round. Because the communication between correct processes is reliable, each correct process will eventually receive those outcomes and proceed to the next round.

The only issue remaining is the possibility of enough processes timing out on every round such that no progress can be made with the remaining processes. Because Turtle Consensus exponentially increases timeouts each round and due to weak synchrony there exist bounds on communication and processing latencies, eventually correct processes will never time out. Given that Turtle Consensus runs for an unbounded number of rounds, there will be an unbounded number of such “good” rounds. □

The previous lemma ensures that if the protocols we use internally in Turtle Consensus satisfy termination when enough correct processes participate, then there will be arbitrarily many rounds when these protocols will run with enough correct processes. Thus eventually all correct processes can make progress and decide.

To satisfy validity, every such decision must come from an input x_i of some process i . This holds due to following two facts: First, any underlying protocol CP selected for a round of Turtle Consensus maintains validity. Second, the inputs of any CP at any round are either directly the inputs of Turtle Consensus (in the first round) or the values contained in the outcomes of the previous round (in all rounds but the first). In Phase 3, if any process i updates its proposal for the next round, it does so either by directly selecting a value from those computed by CP , which by CP 's validity must come from its own inputs, or indirectly using function \mathcal{F}_i . Thus Turtle Consensus satisfies validity.

Finally, we need to show agreement. We observe that Turtle Consensus maintains the following invariant:

Invariant 3. *If $\exists p \in N, v \in V : o_p = (D, v)$ at some round of Turtle Consensus, then the processes with proposal v can never change their proposal in subsequent rounds.*

If $o_p = (D, v)$, then there exists a quorum of processes whose proposals are v (Invariant 1). By Invariant 2 we know that \perp cannot be an outcome for any $j \in N$. Thus, cases 3 and 4 of Phase 3 cannot occur when $o_p = (D, v)$. In addition, if case 2 occurs, then any process, j , that has not crashed or timed-out and has not decided

yet must send (M, v) by definition of outcome (M, v) for $v \in V$. Therefore the only options processes have, are to either keep their old proposal or update it to v .

Using this invariant we can prove the following lemma:

Lemma 5.3.2. *If some correct process $p \in N$ decides $v \in V$ at some round r of Turtle Consensus, then no process $q \in N$ can decide $v' \neq v$ at any round $r' \geq r$.*

Proof. In order for a process to decide some value v at some round r there must be a quorum Q of processes proposing v in that round. If a process decides value v' , $v' \neq v$, in some round r , $r' \geq r$, then another quorum Q' of v' proposals must exist. For $r' = r$ this is impossible because quorums intersect and processes cannot propose multiple values in the same round. For $r' > r$ this means that some correct process with proposal $v \in Q$ must have changed its proposal at some point after round r , because Q and Q' must intersect. By Invariant 3 this cannot happen. \square

Finally, we show that once a decision is made on a value v by some process, the system converges to a state where all processes decide v . This is stated by the following lemma:

Lemma 5.3.3. *If some correct process $p \in N$ decides $v \in V$ at some round r of Turtle Consensus then all correct processes will decide v at some round $r' \geq r$.*

Proof. Let p be the first correct process to decide $v \in V$ at some round r running protocol CP . By Invariant 1 we have that there exists a quorum of processes $Q \in \mathcal{Q}_{CP} : \forall q \in Q : o_q = (D, v) \vee o_q = (M, v)$ in the same round. As a result, in Phase 3 of round r every process will receive at least one of the previous outcomes and thus may change its value to v or keep its previous proposal. Then by correctness of the

CP protocols run at each round and by Lemma 5.3.1 every process will eventually set its proposal to v and decide. \square

In order to show that all correct processes eventually decide, we have to incorporate liveness reasoning of the underlying consensus protocols. For example, if for some protocols rounds terminate with probability larger than 0 and those protocols are invoked an unbounded number of times, then termination is guaranteed with probability 1 [15]. If for some other protocols eventually the timeouts become large enough so that rounds are guaranteed to terminate [38, 64], then this will also guarantee termination for Turtle Consensus.

5.4 Implementation

In this section we describe our implementation of Turtle Consensus. There are four main choices we need to consider:

1. What are the configuration options, that is, what are the contents of \mathcal{P} ?
2. How are configurations selected?
3. How do processes learn the configuration for a round?
4. What is the function \mathcal{F}_i for each process i ?

For the first choice, we used parameterized versions of the single-degree Paxos [64] and Ben-Or [15] protocols. Both protocols can tolerate up to t crash failures with $2t+1$ processes, which is the minimum number of processes required [36]. Nonetheless, these two protocols represent very different approaches to consensus. Simplified, each Paxos round has a single leader that proposes a value that the

remaining processes may accept—if a majority does so the value is decided. The approach deals very well with contention, because a single process selects the next value to try to decide. The approach suffers, however, if the leader fails or becomes slow due to attack.

While in practice multi-decree Paxos is used to prevent having to elect new leaders for each round, our version of single-decree Paxos is parameterized to start with a predetermined active leader selected deterministically as a function of the round number, allowing Paxos Phase 1 to be skipped. If the leader fails early on, the remaining processes time-out and proceed to the next Turtle Consensus round. Thus, in the good scenario, our Paxos implementation will finish in a single phase, that is, a single round-trip message delay.

One issue with this deterministic leader assignment is that a failed process will cause the others to timeout every $|N|$ rounds. To deal with this issue we can use application-specific knowledge associated with the state machine replication protocol. We explain how later.

Ben-Or, on the other hand, is an entirely decentralized approach with no designated leader. Ben-Or uses randomness to resolve conflicts between proposals and offers probabilistic guarantees of termination. Not having a leader, Ben-Or does not suffer from crashed or slow processes, but behaves poorly under contention because it resolves conflicts probabilistically.

To benefit from the different characteristics of the underlying consensus protocols, we need to establish the rules for switching between one another and how to coordinate this switch among all participating processes. In our implementation we keep the approach simple: Switching between the protocols is deterministically

decided as a function of the round number. In this way, any process reaching a particular round r can instantly start Phase 2 and run under the right configuration without need for further communication. Our Turtle Consensus implementation starts from round 0 and runs Paxos at all even rounds and Ben-Or at the odd ones. Moreover, for the Paxos leader rotates as follows: if r is even, then the round is a Paxos round with leader $r/2 \bmod |N|$.

Finally, every process uses the same value-update function \mathcal{F}_{pop} to choose between values in the received outcomes during Phase 3, Case 3: \mathcal{F}_{pop} selects the smallest most popular value from the ones received during Phase 3.

To speed up termination, if a process decides in some round, it immediately broadcasts the decision outcome. An undecided process receiving such a decision outcome decides the respective value. In addition, if a process receives a quorum of outcomes for its current round while in Phase 1 or 2, it will instantly enter Phase 3 with outcome (\top, \perp) (as if it had timed-out).

State Machine Replication

We used Turtle Consensus to implement a basic state machine replication protocol [96]. In this approach, a deterministic state machine implements some service, for example a key/value store. Client requests form the inputs to the state machine, and causes transitions. The results are returned as responses to the requests. A state machine replication protocol (SMRP) uses copies or *replicas* of this state machine, each starting in the same state, and each run by a different process. Each replica is presented with the same sequence of inputs. Because the state machine is deterministic, the replicas will go through the same transitions, producing the

same results, and ending up in the same state. Clients can ignore all but the first response. Therefore, if there are n replicas, $n - 1$ crash failures can be tolerated.

In order to ensure that all replicas receive the same inputs in the same order, we model the input stream as a sequence of slots numbered $0, 1, \dots$ and lazily instantiate Turtle Consensus for each slot. For this, each replica in SMRP maintains a private zero-initialized *instance number*.

Each client connects to a single replica and sends its requests there. A client request contains a client identifier and a request sequence number for that client to form a unique value for each request. Upon arrival of a request from a client, a replica broadcasts the request to the other replicas so that, in the absence of failures, all replicas receive the request. Regardless of the request's origin, a replica that receives a request uses its instance number to instantiate Turtle Consensus locally for that slot using the request as its initial proposal. Depending on the request arrival order, different replicas may propose the same request for different instances of Turtle Consensus. Conversely, each replica may run different instances of Turtle Consensus concurrently for multiple client requests. In order to avoid spawning and managing an excessive number of instances, each replica has a threshold on the maximum number of concurrent instances. Additional requests received are queued until the number of concurrent instances drops below this threshold.

Turtle Consensus messages are tagged with the instance they belong to. When a process receives a message for an instance it has not yet instantiated, the process simply buffers the message until it has been instantiated.

When a Turtle Consensus instance decides some request, the replica logs the decision and notifies the source of the request (either a client or the relaying replica).

At this point the state that the process associates with this Turtle Consensus instance can be garbage collected. If the replica later receives a message for such a garbage collected instance, it can reply to the sender with the outcome of the round as stored in the log. The sender can then decide as well, and likewise garbage collect its state.

Because the same request may be proposed by different replicas in different instances, it is possible that the same client request is decided for multiple slots. Except for the first, the other decisions are treated as no-ops.

Conversely, a request that is proposed in some instance may not be decided by that instance. In order to deal with this issue, each replica maintains a set of all requests that it has proposed but that have not yet been decided. Once there are no outstanding instances, the replica checks to see if this set is empty. If not, the replica creates additional instances. For liveness, clients that connect to crashed replicas should reconnect to another replica and retransmit their outstanding requests.

To deal with the issue caused by deterministic leader assignment for each round, we exploit the configuration window, W , of the SMRP protocol, which determines the number of consensus instances for which there are pending proposals. We attach to each proposal the identifier of the proposing leader. Once the instance decides, the processes learn and log the respective leader's identifier. Processes running Paxos for some instance x can determine the Phase 2 leader by looking up the process whose value was decided in instance $x - W$. If a process fails, it will stop making proposals and its identifier will eventually disappear from the other processes' logs, and thus will not be used as a leader in future instances.

5.5 Evaluation

In this section, we describe how we evaluated Turtle Consensus in the face of various attack scenarios. In Section 5.5.1, we describe our experimental testbed, the attacks, and the parameters of our executions. Then, in Section 5.5.2, we present our findings.

5.5.1 Experiment Setup

We implemented Turtle Consensus and state machine replication in `C++`. Our testbed consists of 5 nodes in Emulab [110], each with 4 cores running at 2.4 GHz, with 12GB of memory. The nodes are connected by 1Gbps switched Ethernet. All communication is over TCP/IP. In our experiments we used $t = 1$. Three of these nodes were designated as the $2t + 1$ participants needed for Turtle Consensus. Of the two remaining nodes, one node is used as attacker, while the other node runs one or more client threads. By running the attacker on a different node, bandwidth attacks do not directly affect the clients' ability to issue requests to the service.

For simplicity of implementation of SMRP we also used 3 replicas, even though technically only $t+1$ are necessary. However, client threads only connect to $t+1 = 2$ of the replicas. This benefits Ben-Or rounds: With only two replicas receiving requests, at most two different values may be used as input for each Turtle Consensus instance, reducing contention compared to using three replicas.

A typical execution of our implementation of SMRP proceeds as follows: Each client thread connects to one of the two replicas, in a round-robin fashion for load balancing purposes, and starts issuing requests. Client threads run in a simple

loop consisting of sending a request and waiting for a response before sending the next request.

The attacker node also consists of a set of threads, which we call *attack threads*. Each of these attack threads targets one replica and sends dummy messages of fixed length as fast as it can. These dummy messages are not requests and are dismissed instantly by the replica. The attack aims to saturate the replica’s bandwidth, disabling them from participating in Turtle Consensus instances if fully successful. The attacker can direct all its threads at a single node or spread them among multiple replicas. Because there is only a single attacker node, the aggregate attack can produce at most 1Gbit/s worth of bandwidth.

In our experiments we measured throughput and latency, both at the client side. Throughput is the aggregate number of operations per second completed by client threads. While SMRP might decide the same request more than once, they are only counted once, thus the throughput we are measuring is not the actual number of instances completed per second on each replica.

By *load* on the service we mean the number of client threads issuing requests. Its value ranges between 2 and 128 client threads spread evenly among the two designated replicas. Each replica has the same number of client threads connected at each run of the experiment. Each run lasted one minute, which we found sufficient to mask any start-up bias that might occur—the performance metrics were not affected substantially in longer runs. In all our experiments, we used 100 byte payloads.

On the attacker’s side the main two parameters are the size of dummy messages sent by each attack thread and the number of attack threads. For the first we picked

200 bytes, which we found sufficient to saturate a 1Gbit link even with a single attack thread. However, we used 8 attacker threads so they could saturate a link to a replica even when sharing it with 64 client threads connected over the same link. Note that 8 attacker threads can send much faster than 64 client threads because the latter are waiting for responses while the former do not.

We used an initial timeout of 1ms for each Turtle Consensus instance, doubling the timeout every round a process times out without having reached a decision. In our experiments we do not consider any server crashes.

5.5.2 Results

We first examine Turtle Consensus when running Paxos on every round. In this Turtle Consensus configuration, the leader is the same in each Turtle Consensus round and instance. We will refer to this configuration of Turtle Consensus as TC-Paxos. In Figure 5.1 we observe aggregate TC-Paxos throughput as a function of the number of client threads under different attack scenarios. The top line shows throughput without any attacks taking place.

The first attack we consider is when an attacker spreads its attack threads evenly among all replicas, saturating approximately 1/3rd of the bandwidth to each replica. This does not affect TC-Paxos performance much because the client requests are relatively small.

The next attack, denoted as “attack on non-active leader” in Figure 5.1 is where the attacker tries to saturate the bandwidth of one of the replicas, but not the node where the leader resides. In each experiment, the attacker picks one of the two remaining nodes at random to attack. In this case, we observe a substantial gap

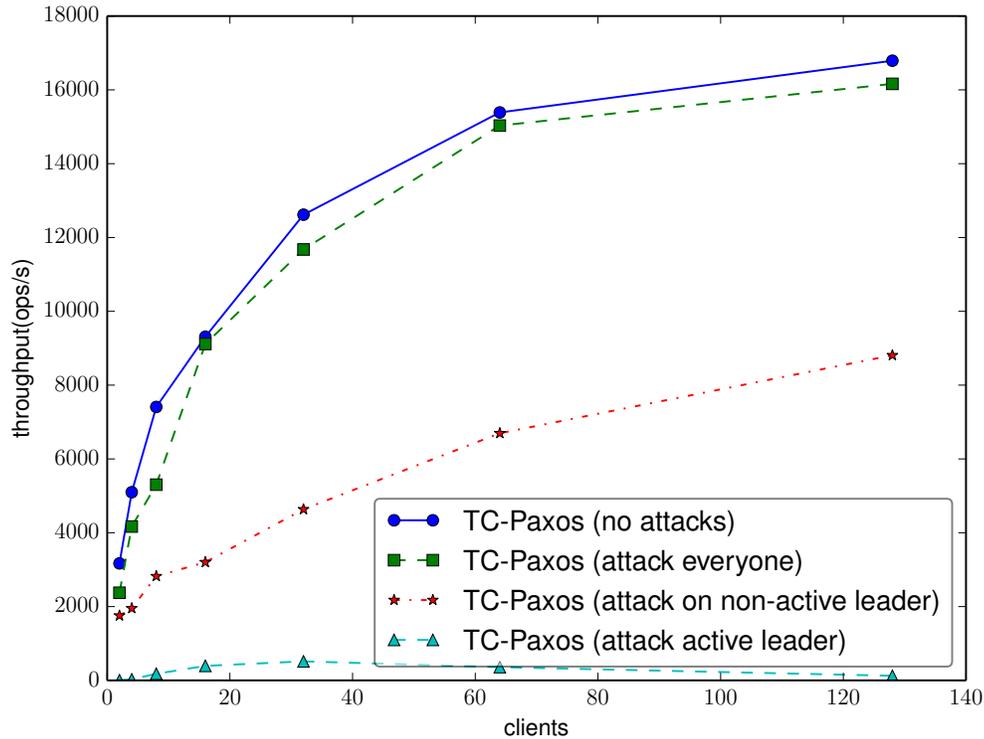


Figure 5.1: Paxos throughput as a function of load and various attack scenarios.

in throughput compared to the case with no attacks. The reason is that when the attacker happens to hit a node that client threads connect to, those client threads (half of all client threads) experience high latency and low throughput during the experiment, bringing the aggregate throughput down substantially.

In the final attack scenario the attacker targets the leader. As seen in Figure 5.1, throughput plummets. This is to be expected: the leader becomes unresponsive, and Turtle Consensus keeps timing out after ever-increasing periods of time. In practice, a Paxos implementation would move the leader role to another replica, and notify the clients (because sending requests directly to the current leader reduces latency). The attacker can exploit this to always know where the current

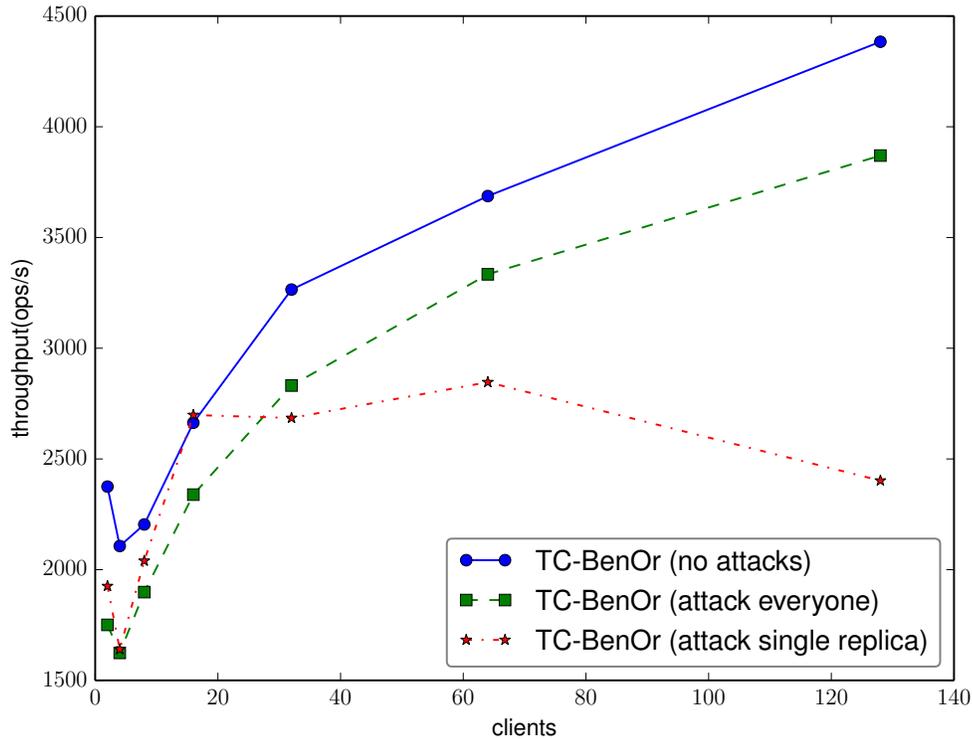


Figure 5.2: TC-BenOr throughput as a function of load and various attack scenarios.

leader resides. Later in this section we will investigate how well the attacker can track the leader.

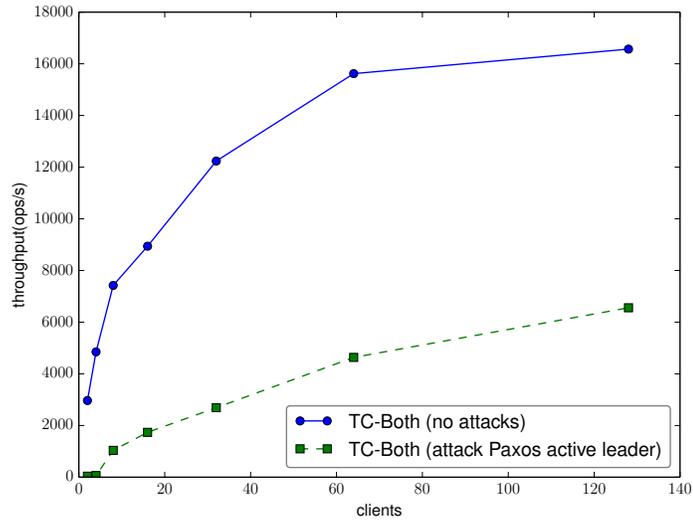
Next we investigate Turtle Consensus when instantiating Ben-Or in each round (Figure 5.2). We refer to this as TC-BenOr. If we compare TC-Paxos with TC-BenOr performance without attacks, then we observe a substantial gap between the two approaches. The leader-based approach of TC-Paxos deals well with contention and usually only requires a single round-trip to decide. However, the probabilistic approach of TC-BenOr often requires multiple rounds to resolve conflicts. Moreover, TC-BenOr’s per-round communication overhead is substantially higher.

We consider two attack scenarios. First, when the attacker attacks every replica equally, TC-BenOr throughput drops similarly to that of Paxos (notice the different scales on Figures 5.1 and 5.2). There is enough bandwidth left over on each replica to give clients satisfactory performance. Next, consider the case where the attacker randomly selects one of the replicas to focus its attack. If a node that client threads connect to is attacked, those client threads witness a significant drop in performance.

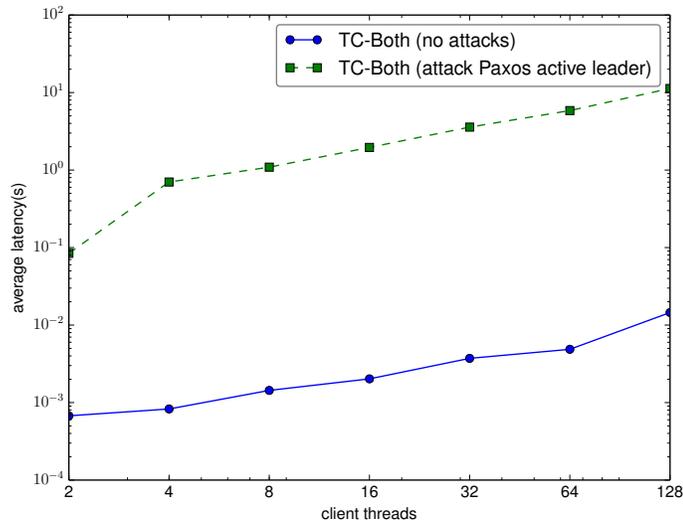
There is, however, another issue as well. With only two effective consensus participants, TC-BenOr takes more rounds to decide than when all three are operational. Recall that client threads connect to only two of the replicas, and thus contention is usually only between two proposals at a time. With three participants, one of the proposals typically gets outvoted in the first round, whereas with two effective participants, decision depends on the outcome of the random choices that the participants make. Thus, under this attack, higher contention results in lower throughput.

Finally, we note that all three cases experience a small drop in performance going from two client threads to four. We believe that this is because at four client threads the likelihood of conflicting proposals in TC-BenOr rounds is higher than with two client threads, which means that TC-BenOr will likely require more than one round to decide. Beyond four client threads, the additional requests increases the number of concurrent TC-BenOr instances, increasing node utilization and thus throughput.

Next we consider the configuration we described in Section 5.4 in which Turtle Consensus runs Paxos at even rounds (starting with the first round) and Ben-Or at odd ones. We denote this configuration TC-Both and show its performance in



(a) Throughput



(b) Latency

Figure 5.3: Turtle Consensus throughput (a) and latency(b) as a function of load alternating between Paxos and Ben-Or with and without attack.

Figure 5.3a. In the case without attack, we see almost the same throughput as TC-Paxos, which is expected because a Paxos round decides on the first round in the absence of failures.

If, however, the leader is attacked, the Paxos round will likely time out (after 1ms) and a Ben-Or round starts. Paxos and Ben-Or rounds keep alternating with timeouts doubling each round. TC-Both provides acceptable performance under high load and attack. TC-Both under attack even outperforms TC-BenOr when not under attack. This can be explained because the attacker targets explicitly the Paxos leader, which happens to be one of the replicas that receives requests. Of the other two replicas, only one receives requests. Therefore the Ben-Or rounds experience little no contention and are likely to decide.

Figure 5.3b shows latency measurements for the same experiment. TC-Both under no attacks shows the latency achieved by our TC-Paxos implementation in the normal case. In the case of an attack on the Paxos leader, the minimum latency starts at around 1ms as expected due to our initial timeout value. Otherwise both latencies grow approximately linear in the number of client threads increases. This is as expected, as the service can only handle a constant number of requests per second, and thus client threads have to await their turns.

Up until now we kept the Paxos leader on the same node, modeling the case where the attacker knows exactly the location of the leader of the current round. In reality, however, rounds do not actually run one at a time and there is some overlap. It may not always be possible for the attacker to know exactly which node to attack if the leader role changes position. In the final experiment, we reconsider TC-Paxos but have its leader rotate among the server nodes on each round. The attacker is now dynamic and tries to follow the leader. Replicas notify clients (both legitimate clients and attackers) where the current leader resides.

Due to our small initial timeout period, however, it is difficult for the attacker to prevent the second round from deciding quickly. Figure 5.4 demonstrates that

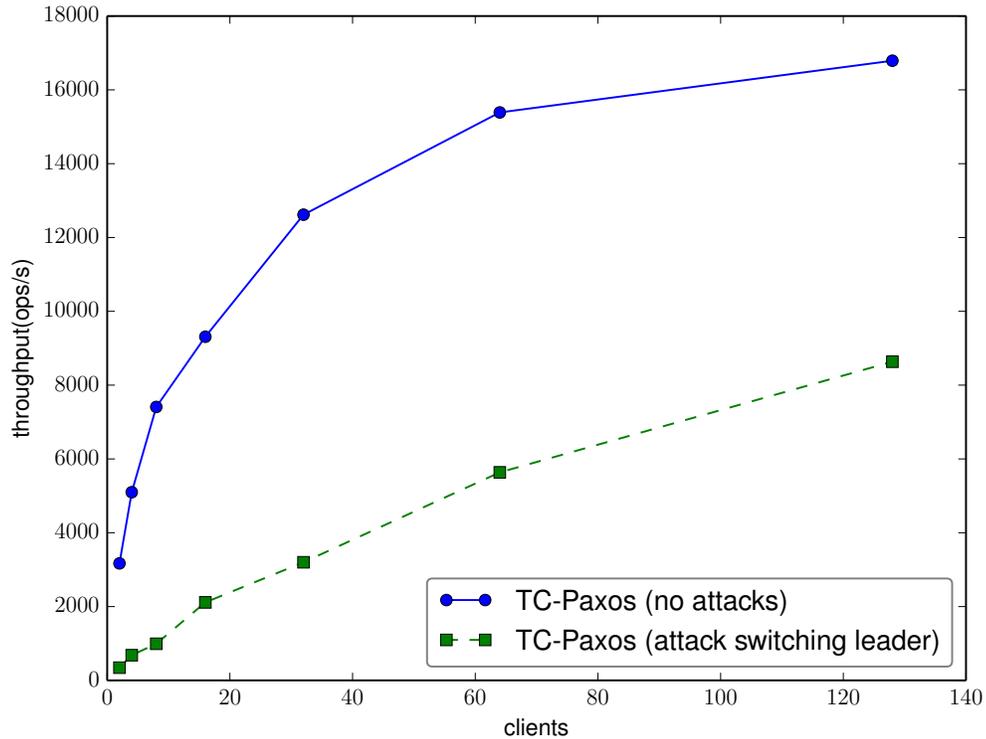


Figure 5.4: Turtle Consensus throughput as a function of load running Paxos with different leader on each round.

the attacker finds out too late that there is a new leader. As part of our ongoing research, we are trying to find ways to make the attacker better at predicting where to focus the attack. At the same time, we are also experimenting with selecting leaders randomly to raise the bar for successful attacks.

Finally, we conducted experiments with larger numbers of replicas (4 and 5). The trends were similar, the main difference being that throughput was lower and average client latency higher. That is to be expected because communication complexity increases for larger number of replicas for both Paxos and Ben-Or.

5.6 Concluding remarks

In this work, we presented a moving target defense approach to deal with Denial-of-Service attacks on a service that uses crash-tolerant consensus. Our protocol, called Turtle Consensus, rapidly changes consensus strategies, which we believe to be important in enhancing a system's resistance to DoS attacks, particularly if they are executed to target specific consensus protocols. We presented our experience with a prototype implementation using two dissimilar underlying consensus protocols: Paxos and Ben-Or, and found that we can achieve both good performance in the absence of attacks and reasonable performance if the system is under attack while also heavily loaded by clients.

Our intention is to consider more sophisticated DoS attacks, while also tolerating Byzantine failures of the replicas themselves. Also, in our current prototype the sequence of underlying consensus protocols is fixed a priori, but we wish to make this dynamic and possibly adapt to any ongoing attacks. We also want to explore the efficiency of our construction, and investigate how we can use Turtle Consensus in larger scale settings by switching not only the protocols but also the set of nodes participating in each round. Finally, we would like to deploy Turtle Consensus in a real service to investigate its effectiveness against DoS attacks.

CHAPTER 6

MOVING PARTICIPANTS TURTLE CONSENSUS

In this chapter we describe an extension to the previously described Turtle Consensus protocol. This extension, that we call Moving Participants Turtle Consensus (MPTC), enables the protocol to not only change the consensus strategy on-the-fly but also the set of processes that execute that strategy. MPTC uses these *moving target defense* strategies to tolerate certain Denial-of-Service (DoS) attacks issued by an adversary capable of compromising a bounded portion of the system. It uses existing cryptographic techniques to ensure that reconfiguration takes place in an unpredictable fashion thus eliminating the adversary’s advantage on predicting protocol and execution-specific information that can be used against the protocol.

We implement MPTC as well as a State Machine Replication protocol and evaluate our design under different attack scenarios. Our evaluation shows that MPTC approximates best case scenario performance even under a well-coordinated DoS attack.

6.1 Introduction

The Turtle Consensus protocol described in Chapter 5, was our first attempt in devising a flexible reactive approach for solving consensus in adversarial environments. It employs fine-grained reconfiguration that significantly changes the attack surface of the protocol, thus making it more robust against protocol targeted attacks. The main strategy of Turtle Consensus is based on using the best approach available for normal operation in a particular setting and switching to different

ones as soon as the approach becomes inefficient, e.g. in the case of a DoS attack. One issue with the previous strategy that became apparent in our evaluation of Turtle Consensus is that this forces the protocol to sub-optimal strategies thus bounding the protocol’s efficiency to the capabilities of these “back-up” protocols. In addition, a more sophisticated adversary capable of compromising even a subset of the system can learn and use the predetermined nature of protocols’ succession to constantly drive the system to sub-optimal executions.

In this chapter we try to address these concerns by adding another degree of freedom in the reconfiguration capabilities of Turtle Consensus. We present Moving Participants Turtle Consensus (MPTC), an extension to the Turtle Consensus protocol that allows switching not only the protocols but also the set of processes on which they run across different rounds of a single consensus instance. The consensus protocol round and the processes participating in its execution form what we call a *configuration*, which our approach changes unpredictably at each round. While the configuration selection for each round is predetermined by a trusted dealer, it is unknown to the processes during MPTC execution. Using existing cryptographic techniques, we ensure that, only if sufficiently many processes collaborate during some round, the next round’s configuration can be determined. This renders MPTC a valuable tool for building systems that can tolerate DoS attacks in both crash- tolerant and byzantine environments where a bounded portion of the system may be compromised.

The chapter is organized as follows: In Section 6.2 we describe the system model and provide some background on the cryptographic primitives we use as well as consensus protocols. In Section 6.3 we present our proposed MPTC protocol for crash failures and demonstrate its correctness. We then extend this description

to address byzantine failures in Section 6.4. Then in Section 6.5 we present an implementation of MPTC and describe how we used it to implemented a state machine replication protocol. We describe our experimental results in Section 6.6. Finally, we make our concluding remarks in Section 6.7.

6.2 Model

6.2.1 Processes and communication

Our system consists of a set of processes \mathcal{N} that communicate using message passing. Each process is modeled as a state machine with a potentially unbounded set of states that executes deterministic or non-deterministic transitions according to some protocol. The protocol specifies the transition function of the processes as well as the messages they exchange. Each process's state consists of two components, the *public* and the *private* or *secret state*. The public state contains the description of the protocol that each process executes and any public cryptographic keys associated with the process. The secret state contains any run-time state the process manages during the execution of the protocol as well as any secret cryptographic keys and/or shares associated with the process. We assume that all keys and shares are stored in a tamper-proof cryptographic coprocessor which executes all operations involving these keys. This ensures the integrity of any public or secret keys used by each process.

Protocol execution and communication are *asynchronous*, meaning that there are no bounds on the time it takes processes to execute transitions and deliver messages.

A process can be correct or faulty. A correct process faithfully executes the protocol and is guaranteed to make progress as long as the conditions specified by the protocol at any given step are eventually met. In this Chapter we will primarily consider crash failures although we extend our techniques to byzantine failures in Section 6.4. A faulty process may crash at any time after which it stops executing the protocol. Up to the point of the crash, honest processes faithfully follow the steps of the protocol. Communication between correct processes is reliable and secure. This means that, in the absence of DoS attack (see below), messages sent by some correct process to another correct process are eventually delivered. It also means that messages are authenticated and cannot be tampered with or fabricated. We assume an upper bound, $f_c < |\mathcal{N}|$, on the total number of processes that might fail during the protocol's execution. In this work we are not concerned with the recovery of failed processes.

Finally, we assume the existence of a special process $T \notin \mathcal{N}$ that from now on we will refer to as *trusted dealer* or simply dealer. The dealer is only used during initialization of the system during which it generates the initial public and secret state of all processes. We assume that during this setup phase the dealer is correct and that it can communicate via secure channels with any process in the system. After initialization, however, the dealer does not execute any protocol steps or exchange messages with any other process.

6.2.2 Adversary and attacks

We assume an adversary, A , that controls which processes fail and when. A is limited on the number of processes it can fail by f_c and cannot fail the dealer. A can also control the delivery order of messages of all processes as well as delay

communication, but must yield to the previously stated reliable communication assumption.

The adversary can also issue *denial of service* (DoS) attacks against the system that can fully saturate the bandwidth resources of at most $f_a < |\mathcal{N}|$ correct processes. This can effectively prevent the targeted processes from progressing in the protocol's execution since they can no longer communicate with the rest of the system. A can change the targets of an attack over time and, in this way, can introduce communication and computation delays on certain processes. The adversary's objective is prevent the system from making progress. From now on we will denote by f the maximum number of processes that can be crashed or under attack during the execution of the protocol, that is $f = f_c + f_a < |\mathcal{N}|$.

In this work, we ignore DoS attacks that target other resources like CPU using legitimate traffic. These attacks can be mitigated using rate limiting techniques such as client cryptographic puzzles [11].

The adversary has read access to the public state of all processes as well as the secret state of up to f processes. We call the processes whose secret state is disclosed to A , *compromised*. While A cannot modify this state it can use it to select the target processes of a DoS attack. Once A has selected the set of compromised processes it can no longer change that set thus preventing A from accessing the secret state of more than f processes. This static approach to compromising processes might seem restrictive and unrealistic. We can make our model more realistic by adopting the approach in [114] where the adversary can change the set of compromised processes over time. This requires defining a window of vulnerability during which the adversary can compromise up to f processes but can change its selection across different windows. We can then use proactive refresh-

ing of cryptographic keys that runs periodically to recover compromised processes across different windows. We will omit this discussion however, since this is not the focus of this work.

Finally, we assume that the various cryptographic schemes we are employing, like public key cryptography and threshold signatures, are secure in the random oracle model.

6.2.3 Cryptographic primitives

Our protocol relies on Threshold Coin Tossing [21]. Here we present a high-level description of this primitive that we will further formalize in Section 6.3. We employ an (n, k, t) threshold coin-tossing scheme in which n parties maintain shares of an unpredictable function, F , mapping an arbitrary bit string, r , to a binary value $\{0, 1\}$. Each of these shares can be used along with an input r to create a value that from now on we will refer to as *function shares*¹. At least k of these function shares of r are required to reconstruct the result $F(r)$, while at most t parties may be compromised. From now on, we will use the term, *function share of $F(r)$* to denote a function share of r generated with a secret share of F .

The scheme defines three functions: 1) The *split* function, which takes as input a function F (represented as a bit string) and creates a set of shares as well as a verification key for each of these shares. 2) The share combining function, *combine*, which takes an input r of F along with k valid function shares of r and produces $F(r)$. 3) The share verification function *verify*, which takes an input r of F , a function share on r and the verification key corresponding to the share that

¹The term used in [21] for these values is coin shares

generated the input function share and determines whether the function share is valid.

This scheme is based on threshold signatures [98] and can be used to create an unpredictable sequence of bits while ensuring that it is computationally infeasible for the adversary to produce an input r and k valid function shares that once combined do not yield $F(r)$. More formally, the scheme satisfies the following properties taken from [21]:

- *Robustness*: It is computationally infeasible for the adversary to produce a value r and k valid shares of r such that the result of the combine function is not $F(r)$.
- *Unpredictability*: Given a value r and functions shares from fewer than $k - t$ correct processes, the adversary can predict the value of $F(r)$ with probability at most $\frac{1}{2} + \epsilon$ for negligible value $\epsilon \in \mathbb{R}$.

The previous unpredictability property was extended to sequences of output bits in [21], such that, given a sequence of values C_i for $i \in \{1, 2, \dots, b\}$ an adversary with fewer than $k - t$ valid shares of some C_i has negligible advantage in predicting $F(C_i)$. From now on, when we talk about unpredictability we will refer to this *extended unpredictability property* of threshold coin-tossing.

Note that the previously described extended unpredictability property allows us to share unpredictable functions in $[\{0, 1\}^* \rightarrow \{0, 1\}^b]$ for any finite b . In other words, we can model each such function as a random number generator that can produce 2^b different values and requires k processes to collaborate in order to produce the random (unpredictable) value corresponding to some arbitrary bit

string r . From now on, whenever we refer to sharing of functions or shares of functions we mean the unpredictable functions that we described in this section.

Threshold coin-tossing can be implemented using any non-interactive threshold signatures scheme that ensures unique valid signature per message as in [98]. A direct implementation of this scheme can be found in [21].

6.2.4 Underlying consensus protocols

MPTC, like other consensus protocols, solves the problem of agreement. In this problem, a set of possibly distributed processes, each of which is initialized with some input value, unanimously and irrevocably output one of those input values. More formally, let \mathcal{N} be a set of processes each of which is initialized with some value from a value set \mathcal{V} . Each process can employ either of the following primitives:

- *propose* a value which allows a process to communicate its value to the rest of the processes in \mathcal{N} ,
- *decide* a value which allows a process to output a value

Every correct consensus protocol must satisfy the following properties:

- *Validity*: If a process decides a value, then that value must be the input value of some process in \mathcal{N} .
- *Agreement*: If any two processes decide they must decide the same value.
- *Termination*: All correct processes eventually decide.

[42] has shown that in an asynchronous environment no consensus protocol exists that satisfies all of the above properties when even only a single failure can occur. To circumvent this result, a variety of protocols have been proposed [15, 29, 4] that use a probabilistic approach and can guarantee the previous properties with the following modification on termination: *All correct processes eventually decide with probability 1*. For the remainder of this work we will refer to the non-probabilistic description of termination as *definite termination* and to the probabilistic one as *probabilistic termination*.

A consensus protocol that implements the previous specification (using either definite or probabilistic termination) even under the presence of t crash failures is called t -crash-resilient. Note that our adversary can additionally perform denial-of-service attacks which can fully saturate a bounded number of processes and render them entirely unavailable. In an asynchronous environment there is no difference between a crashed process and a process that is under DoS attack from the other processes' perspective. For this reason we say that a consensus protocol is correct in our model if it is f -crash-resilient where $f = f_c + f_a$. From now on we will refer to such consensus protocols as f -resilient protocols.

Each process executing MPTC may run different consensus protocols at different rounds. We denote the set of possible protocols each process can choose from by \mathcal{P} . Different consensus protocols make different assumptions under which they are correct, that is meet the previously described specification. The crash-tolerant consensus protocol of Ben-Or [15], for instance, assumes an asynchronous environment and that each infinite schedule has a bounded number of processes performing a finite number of steps. Other protocols make assumptions such as bounds on the number of failures, different degrees of synchrony, the existence of

failure detectors [26], etc. We consider a consensus protocol correct if it satisfies either definite or probabilistic termination. For each protocol $P \in \mathcal{P}$, we denote the set of assumptions required to hold for P to be correct by \mathcal{A}_P . In other words, if assumptions \mathcal{A}_P hold, then P satisfies validity, agreement, and termination. A protocol P is a valid candidate for \mathcal{P} if it is correct under both the assumptions in \mathcal{A}_P and our previous model assumptions regarding failures, network reliability, and adversary.

We only consider consensus protocols operating in rounds and we follow the framework introduced in Chapter 5 for the specification of the round outcomes. According to this specification, every process running a round of a consensus protocol ends up in one of the following states:

$$\{D, U, M\} \times \mathcal{V}$$

where states $(D, v), v \in \mathcal{V}$ indicate that the process has decided v , states $(U, v), v \in \mathcal{V}$ indicate that no process has decided up to the current round, and finally, states $(M, v), v \in \mathcal{V}$ indicate that while the process is not decided, if a decision was made by some process then it must have been v . We will refer to these states as round outcomes or simply *outcomes*. We denote by o_p^r the outcome of process $p \in \mathcal{N}$ at the end of round $r \in \mathbb{N}$.

More formally the following invariants hold about the outcomes of processes completing a round of a correct consensus protocol in \mathcal{P} :

Invariant 4. *If $\exists p \in \mathcal{N}$, $r \in \mathbb{N}$ such that $o_p^r = (D, v)$, where $v \in \mathcal{V}$, then for each correct $q \neq p \in \mathcal{N}$ it holds that $o_q^r = (M, v)$ or $o_q^r = (D, v)$.*

Invariant 5. *If $\exists p \in \mathcal{N}$, $r \in \mathbb{N}$ such that $o_q^r = (U, v)$ for some $v \in \mathcal{V}$ then $\forall q \in \mathcal{N}$, $u \in \mathcal{V}$: $o_q^r \neq (D, u)$.*

This framework facilitates the description of MPTC in the next section and can be used to describe most consensus protocols in literature, including [15, 26, 64, 22].

6.3 Moving Participants Turtle Consensus

In this section we describe our Moving Participants Turtle Consensus (MPTC) protocol. MPTC is an f -resilient consensus protocol operating in rounds such that in each round a different subset of processes may run a different consensus protocol. We start with some preliminary definitions and notation, then describe the protocol, and finally sketch its correctness.

6.3.1 Participants and participant sets

MPTC is run by all processes in \mathcal{N} . In each round, only a subset of \mathcal{N} is actively running a consensus protocol from a set of correct consensus protocols, \mathcal{P} . Let \mathcal{P}_f correspond to the minimum number of processes required to run each protocol in \mathcal{P} . As an example, let \mathcal{P} consist of the Ben-Or [15] and One-Third [28] consensus protocols. The first one requires $2f + 1$ processes to solve the agreement problem tolerating up to f crash failures while the second one needs $3f + 1$. Thus $\mathcal{P}_f = 3f + 1$. We assume that $|\mathcal{N}| \gg f$ and thus $|\mathcal{N}| > \mathcal{P}_f$ for most reasonable f -resilient consensus protocols.

In the remainder of this paper, we say that a process *runs* or *executes* a protocol in \mathcal{P} when it executes a round of that protocol. We will refer to a process executing a protocol in \mathcal{P} at some round of MPTC as a *participant* or an *active participant* of that round. Let $PS = \{S \subseteq \mathcal{N} : |S| = \mathcal{P}_f\}$ be the set of all possible subsets of \mathcal{N} where each subset has size \mathcal{P}_f . We call each such set a *participant set*. A process may be a member of multiple participant sets. In each round r of MPTC, only a single participant set, S_r , is *active*, that is executing a consensus protocol in \mathcal{P} . We assume that participants in each participant set of some round r , $S_r \in PS$, are ordered and denote the i -th participant in S_r as S_r^i . The active participant set for each round is determined by T during initialization, which we describe later in this section.

6.3.2 Configurations

Before describing the initialization procedure and the core of MPTC, we need to define an important concept that encapsulates the information required for a set of processes to run a consensus protocol. We define a *configuration* of MPTC as a tuple $(P, S) \in \mathcal{P} \times PS$. $P \in \mathcal{P}$ describes the consensus protocol to run along with its initialization parameters. To better understand the information contained in the initialization parameters, consider a protocol like Lamport's Paxos [64] and the core consensus protocol he called Synod. In Synod, processes play multiple roles, such as proposers and acceptors. In that sense, P needs to encapsulate not only the protocol under execution, e.g. Synod, but also information related to its initialization such as mapping of proposers and acceptors to processes. The participant set $S \in PS$ corresponds to the set of processes that shall execute the consensus protocol specified by P . Let the set of all possible configurations

$\mathcal{C} = \mathcal{P} \times PS$. Our approach implements a multi-party computation scheme for an unpredictable mapping between natural numbers (rounds) and configurations. We omit details regarding how to represent P since this is an implementation issue and does not affect our protocol. We assume that $|\mathcal{C}|$ is bounded.

6.3.3 Initialization and trusted dealer

We are now ready to describe the initialization of our protocol, how we are using T to create an unpredictable sequence of configurations, and how the active participants of a round can compute the corresponding configuration.

T is a special process that generates the configuration that each process in \mathcal{N} starts with in the first round. It also provides the processes the means to generate configurations for subsequent rounds. To achieve this, T employs a $(\mathcal{P}_f, f + 1, f)$ threshold coin tossing scheme like the one described in Section 6.2.3. Using this scheme, T shares a function F_S between the \mathcal{P}_f processes of each participant set $S \in PS$. Recall that threshold coin-tossing is based on threshold signatures, thus when we say that T shares a function F_S with each participant set, in reality it simply selects a different public-secret key pair for each $S \in PS$ and shares the secret key. Given some round number r , at least $f + 1$ processes in S need to collaborate to produce $F_S(r)$ while up to f of them may get compromised. $f + 1$ is both a sufficient and necessary number of processes to compute the result of the function shared. T cannot be compromised, failed or attacked by the adversary.

At a high-level, T operates as follows:

1. For each $S \in PS$ the dealer picks a function $F_S : \{0, 1\}^* \rightarrow \mathcal{C}$ and generates a secret share, h_S^q , for each $q \in S$.
2. T picks a configuration $C_0 \in \mathcal{C}$.
3. T distributes C_0 and shares to processes over secure channels. $\forall S \in PS$ each process $p \in S$ receives h_S^p and C_0 .

Observe that each function shared by the dealer maps arbitrary strings to configurations. This differs from the functions we defined in Section 6.2.3 which map arbitrary bit strings to bit strings of some finite length b . Since \mathcal{C} is finite, there exists $b = \lceil \log_2 |\mathcal{C}| \rceil$ such that we can trivially obtain an onto function $\{0, 1\}^b \rightarrow \mathcal{C}$. Thus, the functions we need to share can be trivially obtained by the ones supported by the threshold coin-tossing scheme. Note that, by this high-level algorithm, a process in \mathcal{N} will receive multiple shares, one for each participant set it belongs to.

We now discuss how T generates the secret shares. Given model parameters \mathcal{P} and f , we define the *split* function of the threshold coin-tossing scheme presented in Section 6.2.3 as follows:

$$split : [\{0, 1\}^* \rightarrow \mathcal{C}] \rightarrow \mathcal{S}^{\mathcal{P}_f}$$

where $[\{0, 1\}^* \rightarrow \mathcal{C}]$ is the space of all functions from arbitrary bit strings to configurations, and \mathcal{S} is the space of secret shares. *split* breaks the input function into \mathcal{P}_f shares one for each process of some participant set. In other words, $\forall S \in PS$, $split(F_S) = \{h_S^p \mid p \in S\}$. *split* can be implemented using Shamir's secret sharing [97] ($\mathcal{P}_f, f + 1$). Note that in [21] their implementation is based upon verifiable secret sharing since they are considering byzantine failures. In our model,

processes cannot lie and messages cannot be tampered with. As a result, no verification is needed within this context. We extend our approach to byzantine failures in Section 6.4 where we introduce the required verification functionality.

Given a secret share, h_S^p , of some function $F_S : \mathbb{N} \rightarrow \mathcal{C}$ and some input, $r \in \mathbb{N}$, process $p \in S$ can create a function share of $F_S(r)$ using the *share generation* function, $GFS : \mathcal{S} \times \mathbb{N} \rightarrow \mathcal{F}$ where \mathcal{F} is the space of valid function shares that can be generated given a share $h \in \mathcal{S}$ and a natural number. We define, $F_S^p(r) = GFS(h_S^p, r)$. A straightforward implementation of GFS can be derived from the signature share generation for threshold signatures [98].

We define the *combine* functions as:

$$\text{combine} : \mathcal{F}^{f+1} \times \mathbb{N} \rightarrow \mathcal{C}$$

combine works by receiving function shares of some function F_S and some input, r and outputting $F_S(r)$ which corresponds to a configuration. More formally, let

$$F_S^Q(r) = \{F_S^q(r) \in \mathcal{F} \mid \forall q \in Q, Q \subseteq S \text{ and } |Q| = f + 1\}$$

be any set of $f + 1$ function shares of $F_S(r)$, i.e. $F_S^Q(r) \in \mathcal{F}^{f+1}$. Then we have that:

$$\text{combine}(F_S^Q(r), r) = F_S(r)$$

6.3.4 Protocol description

We can now describe the operation of each process executing MPTC. MPTC is an f -resilient round-based consensus protocol in which each round is executed under a different configuration. Let $C_r = (S_r, P_r) \in \mathcal{C}$ be the configuration used for round r , where $S_r \in PS$ is the active participant set and $P_r \in \mathcal{P}$ the consensus protocol specification for that round. Let o_r^p be the outcome of a process $p \in S_r$ running P_r at round r . Let a special value $\perp \notin \mathcal{V} \cup \mathcal{C} \cup \{\{D, M, U\} \times \mathcal{V}\}$ represent the value of an uninitialized variable.

We assume that all processes have common knowledge of \mathcal{N} , f , \mathcal{P} , \mathcal{C} as well as of the functions *GFS combine*. Each process $p \in \mathcal{N}$ runs MPTC with its identifier and some value $x_p \in \mathcal{V}$ as input and at any point in time maintains the following state:

- its current round number, r_p , initialized to 0;
- its proposal $proposal_p$, initialized to x_p ;
- the outcome of a round, o_p representing p 's decision state and initialized to \perp at the beginning of each round; and
- the current configuration c_p describing the currently known active participant set and the consensus protocol the active participants execute; it is initialized to C_0 , which is provided by T during the initialization phase.
- the secret shares, h_S^p , $\forall S \in PS$ such that $p \in S$ provided by T during initialization.

We have organized MPTC description in phases. Messages exchanged between processes carry the number of the phase, the id of the sending process, and the

current round along with the payload. Messages are of the form $\langle \textit{phase number}, \textit{process id}, \textit{round}, \dots \rangle$. Each round, r , of MPTC works in the following 3 phases:

- **Phase 1:** Each process $p \in S_r$ runs a round of the consensus protocol specified by C_r . Let o_p be p 's outcome for round r . If $o_p = (D, v)$, then process p updates $\textit{proposal}_p = v$, decides v and never updates o_p and $\textit{proposal}_p$ again in any future round. If $o_p = (M, v)$, then p updates $\textit{proposal}_p = v$. Regardless of o_p 's value, p goes to Phase 2.
- **Phase 2:**
 - *Step 1:* Each $p \in S_r$ computes function share $F_{S_r}^p(r) = GFS(h_{S_r}^p, r)$ and sends a Phase 2 message $\langle 2, p, r_p, o_p^p, F_{S_r}^p(r) \rangle$ to all processes in S_r . Then p waits for Phase 2 messages from $\mathcal{P}_f - f$ processes in S_r . Once p receives enough messages from some $Q \subseteq S_r$, it proceeds to Step 2.
 - *Step 2:* If $o_p = (U, *)$ where $*$ can be any value in \mathcal{V} , then p updates its proposal to a value v , selected arbitrarily from the outcomes contained in the received Phase 2 messages. It also updates $o_p = (U, v)$.
 - *Step 3:* Let $F_S^Q(r)$ be the set of function shares received from processes in Q . p computes the configuration of the next round, $r + 1$, as $C_{r+1} = \textit{combine}(F_{S_r}^Q(r), r)$ and moves on to Phase 3.
- **Phase 3:** Each $p \in S_r$ sends a Phase 3 message $\langle 3, p, r_p, o_p, C_{r+1} \rangle$ to each process in S_{r+1} . p updates its state: $r_p = r + 1$, $c_p = C_{r+1}$ and if it is still undecided, it updates its outcome, $o_p = \perp$. Each process $q \in S_{r+1}$ that receives Phase 3 messages with the same configuration value, C_{r+1} , from $\mathcal{P}_f - f$ processes, updates its proposal as follows. Let R denote the set of outcomes received:

- Case 1: If $\exists o \in R$ such that $o = (D, v)$, then q updates $proposal_q = v$, decides v , sets its outcome $o_q = (D, v)$ and never updates o_q and $proposal_q$ again in any future round.
- Case 2: If $\forall o \in R$ it holds $o = (M, v)$ for some $v \in \mathcal{V}$ then $proposal_q = v$.
- Case 3: Otherwise, q selects an arbitrary outcome $(*, v) \in R$ where $*$ can be any value in $\{M, U\}$ and updates $proposal_q = v$.

Then q sets $r_q = r + 1$, $c_q = C_{r+1}$ and if it is still undecided, it sets $o_q = \perp$. Finally, it starts the next round.

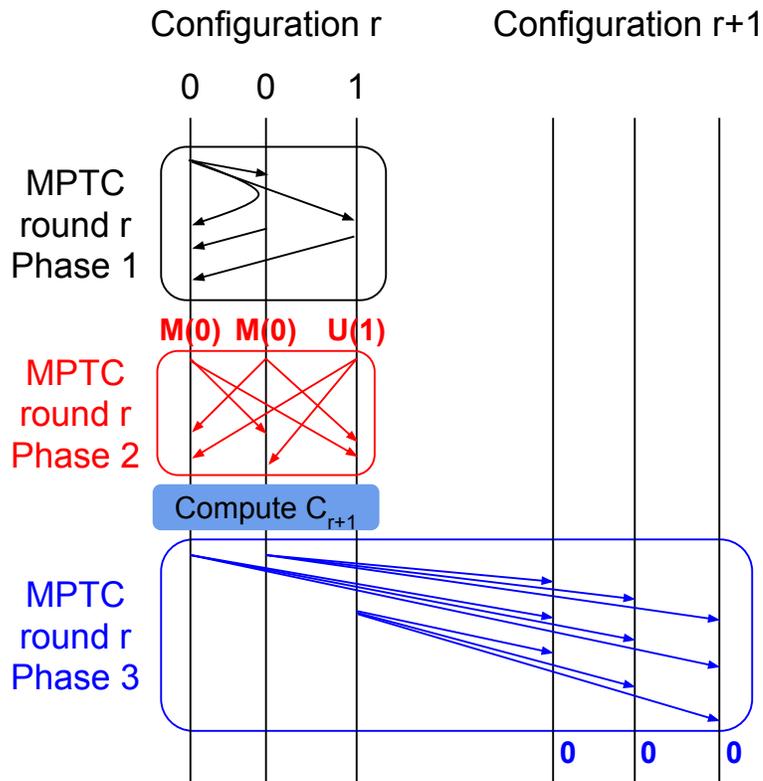


Figure 6.1: A round of MPTC consensus.

Figure 6.1 shows a visualization of the previous round description. MPTC runs for an unbounded number of rounds and eventually reaches a state in which

a decision is made and all correct processes can eventually learn this decision. Messages from old rounds, either delayed in the network or sent by slow processes, are ignored while messages from future rounds are queued to be processed when the receiver reaches that round. Observe that in the previous description, processes in participant sets that never become active take no steps. This will influence the correctness discussion that follows in the next section.

6.3.5 Sketch of Correctness Proof

MPTC needs to satisfy the correctness properties of consensus protocols we defined in Section 6.2. We will show these properties assuming that all the underlying protocols used in \mathcal{P} are f -resilient.

Recall that each protocol $P \in \mathcal{P}$ potentially makes additional assumptions regarding the system model under which the protocol satisfies the previous properties. To guarantee correct execution of P under MPTC, the protocol must be correct under the previous set of assumptions and those assumptions made in Chapter 6.2. Note that this union may contain different assumptions about the same property or aspect of the system. Let P be a consensus protocol for synchronous systems. Then this union of assumptions will contain our assumption that the system is asynchronous and P 's assumption that it is synchronous. In such cases, where one assumption is stronger than another with respect to a particular aspect of the model, we assume that the stronger assumption holds. In our example that would mean that the system would be synchronous.

To facilitate our discussion we will introduce some notation to describe collections of assumptions for the protocols in \mathcal{P} . Let \mathcal{A} denote the set of assumptions

that we made in Section 6.2. For each $P \in \mathcal{P}$, let \mathcal{A}_P denote the set of assumptions P makes about the system and its operation in order to tolerate f crash failures while satisfying the correctness criteria stated in Chapter 6.2. Such assumptions may regard the minimum number of processes required as a function of f , the behavior of the network, the time and ordering of events, restrictions on the adversary, and many more. In this work we will not attempt to accurately model such assumptions since their descriptions vary greatly among different consensus protocols proposed in literature.

Given \mathcal{A} and \mathcal{P} we define the set of assumptions that must hold in every execution of MPTC as

$$\mathcal{A}_{\mathcal{P}} = \left(\bigcup_{P \in \mathcal{P}} \mathcal{A}_P \right) \cup \mathcal{A}$$

In other words, every execution of MPTC must respect the union of all assumptions made by every individual protocol used in Phase 1 as well as our model's assumptions. If an assumption is overridden by another, the stronger of the two holds and thus weaker assumptions are excluded from $\mathcal{A}_{\mathcal{P}}$. Under this definition of $\mathcal{A}_{\mathcal{P}}$ we have that $\forall P \in \mathcal{P}$ every execution of P under $\mathcal{A}_{\mathcal{P}}$, respects the correctness properties of consensus protocols.

We will first discuss validity and agreement and finally we will argue about termination.

Validity. In order to satisfy validity, MPTC needs to ensure that any value decided must be the input value x_p of some process $p \in \mathcal{N}$. This is encapsulated in the following lemma:

Lemma 6.3.1. *If a process in \mathcal{N} running MPTC under $\mathcal{A}_{\mathcal{P}}$, decides a value $v \in \mathcal{V}$, then $\exists p \in \mathcal{N} : x_p = v$.*

Proof. (Sketch) A process running MPTC can decide during Phase 1 or Phase 3. Any decision made during Phase 1 respects validity by correctness of each protocol in \mathcal{P} under $\mathcal{A}_{\mathcal{P}}$. By the outcome and proposal update function of Phases 2 and 3 we described in Section 6.3.4, the processes' proposals and thus possible decision values can only come from the values of the outcomes produced in Phase 1. Since validity holds during Phase 1, if some value v gets decided at some round r , $\exists p \in \mathcal{N}$ whose input value in r is v . \square

From the previous lemma, we have that MPTC initialized with a valid \mathcal{P} satisfies validity under $\mathcal{A}_{\mathcal{P}}$.

Agreement. Agreement is a safety property that ensures that no bad states occur during the system's operation. All correct consensus protocols must satisfy the agreement property for the entirety of their execution. Thus we need to show:

Lemma 6.3.2. *If any two processes running MPTC under $\mathcal{A}_{\mathcal{P}}$ decide values v and v' respectively, then $v = v'$.*

Proof. (Sketch) Assume that $\exists p, p' \in \mathcal{N}$ that decide values $v, v' \in \mathcal{V}$ in rounds $r, r' \in \mathbb{N}$ respectively such that $v \neq v'$. We have the following two cases:

- *Case $r = r'$:* For two different processes to decide different values it must be the case that outcomes $(D, v), (D, v')$ are computed after Phase 1 of r . This cannot occur because each $P \in \mathcal{P}$ is correct under $\mathcal{A}_{\mathcal{P}}$ and therefore respects agreement.

- *Case $r \neq r'$:* W.l.o.g. assume $r < r'$. There must be a round \bar{r} such that $r \leq \bar{r} < r'$ in which all correct processes in $S_{\bar{r}}$ compute either (D, v) or (M, v) outcomes during Phase 1 and after which $\exists q \in S_{\bar{r}+1}$ such that either $o_q = (D, v')$ or $o_q = (M, v')$ and $v' \neq v$. This is impossible: First, Phases 2 and 3 of round \bar{r} only allow processes to change their values to those of the outcomes computed during Phase 1 of \bar{r} , which in our scenario will be v . In other words, all values received by processes in $S_{\bar{r}+1}$ in Phase 3 of round \bar{r} will be v . Second, by the validity property of the protocols in \mathcal{P} no outcome (D, v') or (M, v') can be computed after Phase 1 of round $\bar{r}+1$ if all processes start with proposal v .

By contradiction it must hold $v = v'$. □

Termination. Termination encapsulates the liveness or progress requirement on consensus protocols by ensuring that eventually a decision is or can be made. In our model (Section 6.2), we stated two versions of termination, the definite and the probabilistic one. Notice that by definition, definite termination implies probabilistic termination.

Recall that in MPTC each consensus round is run by a subset of the system's processes. It is possible that in an infinite execution some correct processes may only execute a finite number of consensus protocols rounds and as a result not be able to decide. Note that even if at least one correct process decides, all correct processes can eventually learn this decision by having the decided processes broadcast a special decision message to all processes in \mathcal{N} , which in turn decide upon reception of that message. Therefore to guarantee definite (probabilistic)

termination we simply need to show that eventually at least one correct process in \mathcal{N} decides (with probability 1).

MPTC termination depends on both the guarantees provided by protocols in \mathcal{P} and the properties of the interleavings of rounds of different protocols during MPTC execution. The guarantees of each $P \in \mathcal{P}$ allow us to reason about the properties that each MPTC round satisfies. We know that every round of P must produce an outcome in $\{D, M, U\} \times \mathcal{V}$ provided that \mathcal{A}_P hold. If not, P would violate termination. Note that P should also satisfy termination under \mathcal{A}_P as well, since \mathcal{A}_P makes the same or stronger assumptions. If P guarantees definite termination, then in any infinite execution of P there must be at least one correct process $p \in \mathcal{N}$ that produces outcome (D, v) , $v \in \mathcal{V}$ in infinitely many rounds. If P guarantees probabilistic termination then $\exists \epsilon \in (0, 1]$ such that in any infinite execution of P , infinitely many rounds have probability at least ϵ for at least one correct process to produce outcome (D, v) , $v \in \mathcal{V}$.

Given the previous guarantees provided by the protocols in \mathcal{P} we can show the following:

Lemma 6.3.3. *Every correct process executing a round of MPTC under \mathcal{A}_P eventually completes that round.*

Proof. To show the above we need to ensure that each active participant of some round r completes the all three Phases. Phase 1 completes by the termination property of each protocol in \mathcal{P} with each correct process in S_r computing an outcome with respect to the protocol specified by configuration C_r . Phases 2 and 3 rely on each correct process eventually receiving $\mathcal{P}_f - f$ messages which will occur

due to our assumptions on network reliability and maximum number of failures and processes under attack. \square

Lemma 6.3.3 ensures no process participating in any of the MPTC Phases ever blocks. This *non-blocking* property of MPTC rounds, however, is not sufficient to ensure progress. To reason about progress, we need to reason about the effect of interleaving rounds of different protocols on infinite executions.

To reason about interleavings of consensus protocol rounds we need to reason about configuration sequences and thus about the properties of the unpredictable functions F_S selected by dealer T for each participant set S during initialization. Consider the case where \mathcal{P} contains two artificial protocols P, P' such that any process running P can only decide during an odd round, while any process running P' can only decide on an even round. Assume a pathological infinite execution in which $C_r = (P, S_r)$ if r is even and $C_r = (P', S_r)$ if r is odd. If we can define F_{S_r} for each round r in such a way that the previous configuration sequence is generated during Phase 2 of MPTC, termination cannot be achieved.

More formally, let the set of possible configurations, \mathcal{C} , be based on a valid \mathcal{P} . We define a finite sequence \mathcal{I} of configurations in \mathcal{C} as *conforming* if executing MPTC under $\mathcal{A}_{\mathcal{P}}$ such that \mathcal{I} appears infinitely often, at least one correct process computes a decision outcome in infinitely many rounds. Any finite sequence of configurations in \mathcal{C} that does not have the previous property is called *non-conforming*. The case described in the previous paragraph is an example of such a non-conforming sequence. From now on we will refer to finite sequences of configurations in \mathcal{C} as interleavings.

The previous notion of conforming interleavings raises another restriction on any implementation of MPTC and more specifically on the choice of \mathcal{P} . In every implementation there must be at least one conforming interleaving containing configurations of \mathcal{C} . We call an infinite execution of MPTC in which the corresponding configuration sequence contains infinitely many conforming interleavings, a *conforming execution*.

In the random oracle model, which we assume in this work, the configurations sequences generated by the shared functions F_S have the following property:

Lemma 6.3.4. *Let \mathcal{C} be a set of configurations based on a valid set of protocols \mathcal{P} and let F_S be an unpredictable function for each $S \in PS$ generated by T . Assuming a conforming interleaving \mathcal{I} exists in \mathcal{C} , any infinite sequence of configurations corresponding to an infinite execution of MPTC contains infinitely many occurrences of \mathcal{I} .*

Proof. Each F_S used to generate the next configuration in Phase 2 of each MPTC round is based on the threshold coin-tossing mechanism described in Section 6.2. The unpredictability of this mechanism is based on the use of cryptographic hash functions. In the random oracle model, given some input r these functions produce a value chosen uniformly at random from their co-domain. In other words, at each round r there is a positive probability for each configuration $C \in \mathcal{C}$ to be selected as the next configuration to run. Therefore in an infinite execution, any interleaving in \mathcal{C} appears infinitely often. Thus conforming interleaving \mathcal{I} appears infinitely often. □

By the previous Lemma we have that:

Corollary 1. *Assuming a conforming interleaving \mathcal{I} exists in \mathcal{C} , any infinite execution of MPTC is conforming.*

We can now reason about the termination guarantees of MPTC, under the assumptions that \mathcal{P} is valid, that there is a conforming interleaving in \mathcal{C} and that $\mathcal{A}_{\mathcal{P}}$ hold during MPTC's execution. Depending on the protocols executed under the configurations of a conforming interleaving, definite or probabilistic termination can be guaranteed. This is shown in the following lemmas:

Lemma 6.3.5. *Let \mathcal{I} be a conforming interleaving in \mathcal{C} that contains at least one round of a protocol satisfying definite termination. In any infinite execution of MPTC there is at least one correct process that makes a decision.*

Proof. By Lemma 6.3.4 we have that \mathcal{I} will be executed infinitely often and so will the round of some protocol P satisfying definite termination. By the termination guarantees of P it must be the case that at least one correct process computes a decision outcome (D, v) for some $v \in \mathcal{V}$ in infinitely many rounds. Therefore eventually at least one correct process running MPTC decides. \square

We can state a similar lemma for probabilistic termination:

Lemma 6.3.6. *Let \mathcal{I} be a conforming interleaving in \mathcal{C} that contains rounds of protocols satisfying probabilistic termination. In any infinite execution of MPTC there is at least one correct process that makes a decision with probability 1.*

Proof. The proof is similar to that of Lemma 6.3.5 except from the fact that the rounds in which a decision can be made can only ensure that at least one process decides with probability 1. \square

Finally, we need to show the security properties satisfied by MPTC under the previous assumptions. More specifically we need to show the following two properties:

Lemma 6.3.7. *Robustness: It is computationally infeasible for A to produce r and $f + 1$ shares of r for any participant set S such that the output of combine given the previous shares and value as input is $F_S(r)$.*

Proof. It follows directly from the robustness property of the threshold coin-tossing scheme we use for computing the next round's configuration in each MPTC round. The property is proven in [21]. \square

Lemma 6.3.8. *Unpredictability: Let C_r^A be A 's prediction for $F_{S_r}(r)$ after learning at most f function shares for $F_{S_r}(r)$ as well as any number of functions shares for $F_{S_{r'}}(r')$ for arbitrary many $r' < r$. Then the probability of $C_r^A = F_{S_r}(r)$ is at most $\frac{1}{|\mathcal{C}|} + \epsilon$ where $\epsilon \in \mathbb{R}$ is negligible.*

Proof. This property follows from the implementation of each F_S as a $(\mathcal{P}_f, f + 1, f)$ threshold coin-tossing scheme and from the extended unpredictability property of a sequence of coins produced by this scheme shown in [21]. By selecting the length of the sequence to be $m = \lceil \log |\mathcal{C}| \rceil$ we ensure that the probability of A predicting the next configuration having compromised at most f processes in the active participant set is $\frac{1}{2^m} + \epsilon \leq \frac{1}{|\mathcal{C}|} + \epsilon$ for negligible security parameter ϵ . \square

The previous termination discussion relies on the random oracle assumption we made earlier. While this assumption is important for supporting the unpredictability and robustness properties of our configuration generation scheme it is not necessary if such properties are not needed. If we wanted to drop this assumption, we would need to place additional restrictions on \mathcal{P} and \mathcal{C} to satisfy

termination for MPTC. More specifically, we need to ensure that any interleaving in \mathcal{C} is conforming. Under that assumption the termination results still hold since any infinite execution consists of conforming interleavings in which at least some correct process makes a decision.

6.4 Extension to Byzantine Failures

Byzantine agreement model

Our MPTC protocol can be extended to support byzantine agreement. Under this new adversary assumption, we call a process honest if it faithfully executes the protocol. An honest process may crash during the execution but up to the point of crash its execution does not deviate from the protocol description. Observe that in our previous model (Section 6.2) all processes were honest. We call a process correct if it is honest and eventually makes progress. This implies that a correct process never crashes. Faulty processes, on the other hand, may deviate arbitrarily from the protocol but cannot alter the secret state.

The specification of the protocols in \mathcal{P} also change. Each protocol in \mathcal{P} is now a *Byzantine Fault-Tolerant* (BFT) agreement protocol. In this problem, the objective is for all honest processes to agree on the same value. The correctness criteria are as follows:

- *Agreement*: If two honest processes decide, they decide the same value.
- *Validity*: If an honest process decides value v , then v was proposed by at least some process.

- *Unanimity*: If all honest processes propose the same value v , then an honest process that decides must decide v .
- *Termination (Definite)*: All correct processes must eventually decide.
- *Termination (Probabilistic)*: All correct processes must eventually decide with probability 1.

Note that it now holds $\mathcal{P}_f \geq 3f + 1$. Also note that the byzantine failures assumption is now part of our model assumptions set \mathcal{A} and thus is included in $\mathcal{A}_{\mathcal{P}}$ for any set of BFT protocols \mathcal{P} . The round outcomes framework described in Section 6.2 still hold under the following modifications on invariants 4 and 5:

Invariant 6. *If there exists honest $p \in \mathcal{N}$, $r \in \mathbb{N}$ such that $o_p^r = (D, v)$ where $v \in \mathcal{V}$ then for each correct $q \neq p \in \mathcal{N}$ it holds that $o_q^r = (M, v)$ or $o_q^r = (D, v)$.*

Invariant 7. *If there exists honest $p \in \mathcal{N}$, $r \in \mathbb{N}$ such that $o_q^r = (U, v)$ for some $v \in \mathcal{V}$ then there is no honest $q \in \mathcal{N}$, $u \in \mathcal{V}$ such that $o_q^r = (D, u)$ and $u \neq v$.*

This new version of invariants refer to honest processes since faulty processes may update their state arbitrarily at any point in time. Thus the processes' outcomes are meaningful only for honest and correct processes.

Trusted dealer protocol

Let \mathcal{P} be a set of BFT protocols that tolerate up to f failures. The trusted dealer initialization protocol now becomes:

1. The dealer assigns an identity for each $p \in \mathcal{N}$ using a public - key cryptography scheme. It generates a public-private key pair, $(public_p, private_p)$ for each $p \in \mathcal{N}$.

2. For each $S \in PS$ the dealer picks a function $F_S : \mathbb{N} \rightarrow \mathcal{C}$ and generates a verification key VK_S and for each $q \in S$ a secret share, h_S^q and a verification key VK_S^q .
3. T distributes shares and keys to processes over secure channels. $\forall S \in PS$ each process $p \in S$ receives h_S^p , VK_S , and VK_S^q , $\forall q \in S$. In addition, each process p receives $(public_p, private_p)$ and the public keys of all other processes.

In the byzantine case, we need to use a verifiable secret sharing scheme like [40, 82] since we need a way to ensure that invalid function shares created by faulty processes can be identified and discarded by honest ones. In such schemes, a verification function is specified which typically works by receiving a value r , a share of $F_S(r)$ and some verification keys that depend on F_S and the secret share used to generate the previous function share and outputs 1 or 0 indicating whether the share provided is a valid share of $F_S(r)$ or not. We define our verification function, *verify*, as follows:

$$verify : \mathbb{N} \times \mathcal{F} \times \mathcal{K}^2 \rightarrow \{0, 1\}$$

where \mathcal{K} is the space of verification keys. Note that the *verify* above is modeled after the share verification algorithm presented in [21]. Given a function share $F_S^p(r)$, it receives 2 verification keys, VK_S , and $VK_S^p \in \mathcal{K}$, the first produced using F_S and the second using h_S^p .

T can generate these verification keys using a secure cryptographic hash function, that is a function that is easy to compute but computationally infeasible to reverse. Feldman [40] provided some example functions with this property, with RSA being one of them. We abstract away such details and denote by $hash : \{0, 1\}^* \rightarrow \mathcal{K}$

a function that has this property. T can use $hash$ to generate the above mentioned verification keys during phase 1 (after using split) as follows:

$$VK_S = hash(F_S), \forall S \in PS$$

$$VK_S^p = hash(h_S^p) \forall p \in S, \forall S \in PS$$

Note that our $hash$ function takes different types of input, such as $\{0, 1\}^* \rightarrow \mathcal{C}$ and \mathcal{S} , but both are bit strings and so are the elements of \mathcal{K} and \mathcal{F} . While we are using different domain notations to distinguish between functions, secret and function shares, and verification keys, any implementation of these schemes is working with bit strings.

Byzantine Tolerant MPTC

The Byzantine version of our MPTC protocol is similar to that in Section 6.3.4. The processes maintain the same state as in Section 6.3.4 plus the cryptographic keys generated by the dealer. All messages are signed using these keys. Messages are of the form $\langle Phase\ number, process\ id, round, signature, \dots \rangle$ and processes ignore messages with invalid signatures or messages not destined to them. The protocol operates in rounds, the configuration of the first round, C_0 , is determined by T and is known by all processes and each round proceeds as follows:

- **Phase 1:** Each process $p \in S_r$ runs a round of the BFT protocol specified by C_r . Let o_p be p 's outcome for round r . If $o_p = (D, v)$, then process p updates $proposal_p = v$, decides v and never updates o_p and $proposal_p$ again in

any future round. If $o_p = (M, v)$, then p updates $proposal_p = v$. Regardless of o_p 's value, p goes to Phase 2.

- **Phase 2:**

- *Step 1:* Each $p \in S_r$ computes function share $F_{S_r}^p(r) = GFS(h_{S_r}^p, r)$ and sends a Phase 2 message $m = \langle 2, p, r_p, o_p^p, sign_m, F_{S_r}^p(r) \rangle$ to all processes in S_r , where $sign_m$ is the signature of message m . Then p waits for $\mathcal{P}_f - f$ Phase 2 messages with valid function shares from processes in S_r . Each process receiving a Phase 2 message checks the validity of the function share contained within using $verify(r, F_{S_r}^p(r), VK_S, VK_S^p)$. Once p receives enough such messages from some $Q \subseteq S_r$, it proceeds to Step 2.
- *Step 2:* If $o_p = (U, *)$ where $*$ can be any value in \mathcal{V} , then p updates its proposal to a value v , selected arbitrarily from the outcomes contained in the received Phase 2 messages. It also updates $o_p = (U, v)$.
- *Step 3:* Let $F_S^Q(r)$ be the set of valid function shares received by processes in Q . p computes the configuration of the next round, $r + 1$, as $C_{r+1} = combine(F_S^Q(r), r)$ and moves on to Phase 3.

- **Phase 3:** Each $p \in S_r$ sends a Phase 3 message $m = \langle 3, p, r_p, o_p, sign_m, C_{r+1} \rangle$ to each process in S_{r+1} . p update its state: $r_p = r + 1$, $c_p = C_{r+1}$ and if it is still undecided, it updates its outcome $o_p = \perp$. Each process $q \in S_{r+1}$ that receives Phase 3 messages with the same configuration value, C_{r+1} , from $\mathcal{P}_f - f$ processes, updates its proposal as described below. Let R denote the set of outcomes received:

- *Case 1:* If there are at least $f + 1$ outcomes in R of the form (D, v) for some value $v \in \mathcal{V}$, then process q updates $proposal_q = v$, decides v , sets

its outcome $o_q = (D, v)$ and never updates o_q and $proposal_q$ again in any future round.

- Case 2: If $\exists v \in \mathcal{V}$ such that at least $f + 1$ outcomes in R are (D, v) or (M, v) then $proposal_q = v$.
- Case 3: Otherwise, q selects an outcome $(*, v) \in R$ where $*$ can be any value in $\{M, U\}$ and v is the most frequent value in R , breaking ties arbitrarily. q then updates $proposal_p = v$.

Then q sets $r_q = r + 1$, $c_q = C_{r+1}$ and if it is still undecided, it sets $o_q = \perp$. Finally, it starts the next round.

The previous protocol runs for an unbounded number of rounds, like our main MPTC protocol, and eventually reaches a state in which at least $f + 1$ correct processes decide and thus all correct processes can learn that decision.

Correctness discussion

The byzantine version of MPTC must satisfy the properties described above as well as robustness and unpredictability. We again assume a valid \mathcal{P} containing correct BFT protocols, MPTC execution under $\mathcal{A}_{\mathcal{P}}$ as well as a set of configurations \mathcal{C} such that a conforming interleaving exists. The arguments for all of these properties are similar to those in Section 6.3.5. In fact, the termination arguments for the byzantine MPTC are exactly the same. Robustness and unpredictability are also exactly the same and our arguments are mainly borrowed from the corresponding arguments of the byzantine agreement protocol of [21]. For the rest of the properties we have the following results:

Lemma 6.4.1. *Byzantine MPTC satisfies validity.*

Proof. By correctness of the underlying BFT, any decision made by an honest process during Phase 1 respects validity. A decision during Phase 3 can only occur if at least $f + 1$ processes send the same decision outcome (D, v) for some $v \in \mathcal{V}$. For that to happen at least one honest process must have computed that outcome during Phase 1. Thus that value must have been proposed by some process. \square

Lemma 6.4.2. *Byzantine MPTC satisfies agreement.*

Proof. Note that Phase 1 respects agreement by the correctness of the protocols in \mathcal{P} . Observe that honest processes can only decide during Phase 3 if at least $f + 1$ processes have decided the same value. This means that if an honest process decides v during Phase 3 of MPTC at least one honest process has already decided v . Using similar arguments to those in Lemma 6.3.2, we can show that at any point during MPTC execution any two honest processes that decide must decide the same value. \square

Lemma 6.4.3. *Byzantine MPTC satisfies unanimity.*

Proof. Assume that all honest processes are initialized with the same value $v \in \mathcal{V}$. Since the BFT protocols in \mathcal{P} are correct, honest processes getting into Phase 1 with the same value, v , can only compute outcomes (D, v) , (M, v) and (U, v) . Otherwise it would mean that honest processes can change their proposals between rounds of the BFT protocol in question (e.g., via the influence of the faulty processes) and eventually decide a different value, which would violate unanimity of the BFT protocol. Therefore, at the end of Phase 1 all honest processes will end up with outcomes containing v . v will be the most frequent value in any subset of $\mathcal{P}_f - f$ outcomes since $\mathcal{P}_f \geq 3f + 1$ and at least $2f + 1$ processes are correct. As a result Phases 2 and 3 will have all honest processes updating their outcomes

and proposals to v . Consequently, all honest processes will eventually move to subsequent rounds with v as their proposal and thus v will be the only value that can be decided by some honest process. \square

6.5 Implementation

In this section, we describe a simple implementation of a non-byzantine version of MPTC as well as a state machine replication protocol we built on top of it. To implement MPTC we need to decide on the following parameters:

- The choice of protocol set \mathcal{P} .
- The set of possible configurations \mathcal{C} , which specifies not only the protocol of each round but also its initialization.
- The configuration selection functions $F_S, \forall S \in PS$, generated by the trusted dealer.
- The implementation of the *split* function.
- The implementation of *GFS* and *combine* used during the main protocol.

Our set of protocols, \mathcal{P} , contains only a single consensus protocol, an optimized version of single decree Paxos [64]. A round of single decree Paxos operates in 2 phases. In the first phase an active leader/proposer gets elected and in the second the active leader makes its proposal to the rest of the processes who may accept the proposal. If the proposal gets accepted by a majority of acceptors, it gets decided. Paxos tolerates f crash failures using $2f + 1$ processes and thus $\mathcal{P}_f = 2f + 1$.

Similar to the implementation of Turtle Consensus [79], we avoid electing leaders in each round by using a parameterized version of single decree Paxos in which each round comes with a predetermined leader known to all active participants. Assuming an ordering of the processes executing the protocol, we can set the leader to be the process whose position in that order is equal to round number modulo $2f + 1$. Under normal execution conditions, the previous optimization yields the following benefits: 1) a decision within a single round-trip of communication since it directly executes Phase 2 of Paxos, and 2) performance unaffected by contention since the active leader is the only proposer.

We assume the weakest failure detector, $\diamond\mathcal{W}$, presented in [25] in order to deal with failures. If a leader failure is suspected by $\diamond\mathcal{W}$, then the suspecting processes will complete that round using M or U outcomes depending on whether they have received a proposal or not, respectively. We implement $\diamond\mathcal{W}$ like we did in Turtle Consensus, using timeouts with exponentially increasing timeout periods when processes are inaccurately suspected. This way we ensure that there will be enough rounds executed “concurrently” by sufficiently many processes which is critical for ensuring termination in our Paxos variant.

In the description above we did not specify how the states of processes running Paxos are turned into outcomes at the end of a Paxos round. The process executing as the active leader can either decide the value it proposes during Phase 2, say v , or fail to do so due to either failing or suspecting a majority of acceptors. In the first case, it computes outcome (D, v) . In the second case and if the leader did not fail, it will timeout knowing that if a decision was made it must have been for its own proposal, thus computing outcome (M, v) , where v is its proposal in the beginning of the round. Similarly, a process running as an acceptor can end a round either

having accepted the active leader’s proposal, thus computing outcome (M, v) , or timing out without having received any proposal, in which case it does not know anything about the decision progress. In this latter case, we need to ensure that if the acceptor becomes the next round’s active leader, it will make a proposal consistent with already accepted proposals. The way original Paxos achieves this, is through its phase 1. As a result our variant requires processes exiting the Paxos round without knowledge of the round’s decision state to retrieve this knowledge from the rest of the processes.

To avoid incurring another round of communication in our Paxos variant, we piggyback this decision state retrieval onto Phase 2 of MPTC. Timed out processes can use the set of outcomes received to update their proposal. The update procedure is similar to the one used by processes that have computed outcome $(U, *)$. The main difference is that processes without knowledge about the outcome of the round send a special “unknown” outcome. These “unknown” outcomes are ignored by receiving processes unless all outcomes received during Phase 2 of MPTC are “unknown”, in which case no decision could have been made. In the latter case, the receiving process updates its outcome to (U, v') , where v' is the receiver’s proposal at the beginning of the round, and acts as in Phase 2, Step 2, Case 3. The above optimization ensures that if the predetermined leader decides value v at some round r and a failure prevents that decision to be learnt in r , then all processes during Phase 2 of r will receive at least one (M, v) outcome. The receiving processes will be forced to adopt v and thus future proposals can only be about v .

Our set of configurations is $\mathcal{C} = \{(S, P) | S \in PS\}$ where $P \in \mathcal{P}$ is the described Paxos variant. We assume that for every participant set there is an ordering of the processes in it. This is easy to achieve using the processes unique identifiers,

and it facilitates the selection of leader of each configuration without having to define additional initialization information in each configuration. Note that since all rounds execute the same correct consensus protocol, all possible interleavings in \mathcal{C} are conforming.

Observe that in contrast to our prior work on Turtle Consensus we use the same protocol across configurations. In Turtle Consensus (Chapter 5), different configurations used the same $2f + 1$ set of processes. As a result, the adversary could try to track the current leader within that set of processes even if the leader changed across different configurations. Therefore, a competent adversary could eventually locate and force Turtle Consensus rounds to fail, which can lead to poor performance. For that reason, we kept switching between a leader-based (Paxos) and fully decentralized (Ben-Or) consensus protocols across configurations to prevent the adversary from exploiting the leader vulnerability. A side-effect of that approach, however, was that by falling back to a less efficient protocol (Ben-Or) we only achieved sub-par performance compared to the graceful execution using only Paxos rounds. With MPTC we do not need to employ such tactics since the adversary now needs to scan through $|\mathcal{N}| \gg f$ processes before it can identify the leader of our Paxos configuration.

The remaining parameters of our implementation are related to the cryptographic framework assumed by our protocol. While we did not implement this framework for our evaluation, we describe for completeness how we could do so in the following paragraphs. For the actual implementation that we evaluate in Section 6.6, we assume that all participant sets use the same unpredictable function given to all processes via a configuration file. This file defines a sequence of configurations, one for each round, that processes move to in a round-robin fashion,

that is use the first configuration in round 0, the second in round 1, etc. Once the last configuration in the sequence is used, the processes loop back to the first one and continue from there. We emulate the restrictions that the cryptographic framework imposes on the adversary by assuming that only the processes in S_r and S_{r+1} can learn C_{r+1} and only after Phase 2 of round r completes.

A potential implementation for the unpredictable functions is having $F_S = F$, $\forall S \in PS$, where F is derived from the threshold coin-tossing scheme implementation based on Diffie-Hellman and presented in [21]. This approach hashes the input value, r , using a cryptographically secure hash function, modeled as a random oracle, and raises the result to a secret exponent. This exponent is shared among the processes using Shamir's secret sharing [97]. Finally, the result is further hashed to obtain the value of $F(r)$ using another cryptographically secure hash function.

Function *split* used by the dealer during initialization can be implemented using Shamir's secret sharing as mentioned above. In the case of byzantine failures, a verifiable secret sharing scheme like Feldman's [40] would be required in order to implement both *split* and *verify*. Function *GFS* can be implemented by having each process p hashing the input round number r and raising it to its secret share of the exponent it received from the dealer. Finally, *combine* for (byzantine) MPTC simply multiplies a set of $f + 1$ distinct (valid) shares and hashes the result. Note that the *combine* computation is slightly more complicated and a detailed version of its implementation can be found in [21]. Also note that one can alternatively use any non-interactive threshold-signature scheme with the property that there is only one valid signature per message, like the RSA-based scheme of Shoup [98]. We can then obtain the value of the function by hashing the resulting signature computed by $f + 1$ signature shares on a message containing input r .

6.5.1 MPTC-based state machine replication

We used the previous implementation of MPTC to build a SMRP, similar to the one described in Section 5.4. While the components of the implementation are similar, their interactions are different. There are three sets of processes, the clients, the replicas \mathcal{R} , and the participants \mathcal{N} . The clients issue requests to the participants who order these requests and forward them to replicas. Replicas execute the received requests in the order established by participants and send the results back to participants who then forward them back to clients.

Messages exchanged in this implementation are of the form

$$\langle type, src, dst, (content-attribute-1, content-attribute-2, \dots) \rangle$$

where $type \in \{\text{REQUEST}, \text{RESPONSE}, \text{DECISION}, \text{RECONFIGURATION}\}$, $src, dst \in \mathbb{N}$ are the ids of the communicating processes, and $content-attribute-x$, for $x \in \mathbb{N}$ constitute the message payload. The different types of messages are as follows: A *request* message is sent by a client to a participant and may be forwarded between participants. It carries a deterministic operation to be executed by the service. A *response* message is sent by a replica to a participant which then forwards it to clients and contains the result of the execution of an operation issued via a request by the client. A *decision* message is sent by a participant to a replica, and carries a request along with its order of execution. Finally, *reconfiguration* messages are sent between participants and are used to update the configuration of the MPTC execution.

Clients

Clients are uniquely identified processes whose identities are independent from those of the participants and the replicas. They connect to at least $f + 1$ participants to which they send requests of the form:

$$(client-id, request-number, command)$$

where *request-number* is a unique identifier for a particular request sent by this client and *command* is an application-specific description of a deterministic operation and of any arguments that operation requires. Note that each pair $(client-id, request-number)$ uniquely identifies a request received by the state machine and will be used by participants and replicas to track requests that are new, under processing, or executed. Clients therefore maintain the following state:

- $cid \in \mathbb{N}$, which is initialized with a unique value identifying the client.
- $rsn \in \mathbb{N}$, which is initialized to 0, incremented each time the client issues a new request and uniquely identifies requests for a given client.
- $pending-requests \subseteq \mathbb{N}$, which is an initially empty set that stores ids of requests the client has sent, but has not yet received response.

Our clients are modeled as state machines with the following transitions:

T1: Precondition: There is a command *cmd* that needs to be executed

Action: Send message $\langle cid, pid, REQUEST, (cid, rsn, cmd) \rangle$ to each participant *pid* that *cid* is connected to; add *rsn* to *pending-requests* and update $rsn = rsn + 1$.

T2: **Precondition:** Received message $\langle pid, cid, \text{RESPONSE}, (rsn', result) \rangle$

Action: If $rsn' \notin \text{pending-requests}$ ignore; otherwise remove rsn' from *pending-requests*.

Participants

The participants receive requests from clients and are responsible for ordering these requests and send them for execution to the replicas. To achieve this they spawn MPTC instances, one for each request that needs to be ordered. Each instance has its own identifier and decided requests are ordered according to the identifiers of the MPTC instances that decided them. Only one participant set can be active at any point in time. The participants of this set are responsible for creating consensus instances. The active participant set may concurrently run different instances of MPTC to deal with multiple requests. As in the case of Turtle Consensus [79], we consider a global threshold W that limits the number of concurrent consensus instances that have not yet decided. Requests that arrive when W concurrent MPTC instances are running are queued and processed when some of the running instances complete.

Note that in most common SMRPs clients typically send their requests to replicas directly, which are responsible for instantiating consensus instances to these requests. By disabling communication between clients and replicas we prevent clients from launching DoS attacks on the replicas. This approach does however have the issue that when a decision is made and the corresponding request is executed, two hops of communication are needed for the response to arrive at the clients. In addition, it burdens participants with implementing functionality beyond running consensus that is commonly offered by replicas, like keeping track of

the values already decided, maintain state for ongoing instances and other aspects of the implementation, which makes it a less efficient approach. In this implementation, however, we are primarily interested in building an attack-tolerant SMRP and thus made the choice of communication pattern we described above.

Each participant is connected to all replicas and zero or more clients. Apart from the state required to run MPTC described in Section 6.3, each participant additionally maintains the following state:

- $pid \in \mathcal{N}$, initialized with the identifier of the participant.
- $next-instance \in \mathbb{N}$, initialized to 0, stores the id of the next instance that will be created.
- $configuration \in \mathcal{R}$, initialized to C_0 provided by the dealer, stores the configuration that pid considers active.
- $requests \subseteq \mathbb{N}^2 \times Ops$, which is an initially empty set that stores requests that have been received but not yet decided. Ops denotes the space of commands and it is application-specific.
- $instances \subseteq \mathbb{N}^3$, which is an initially empty set that stores for each running instance the id of the instance as well as the request identifier (cid' , rsn); the request identifier is pid 's input proposal for that MPTC instance.
- $rstate \in \{\text{TRUE}, \text{FALSE}\}$, initialized to **FALSE**, indicates whether the participant is currently under reconfiguration.
- $responses \subseteq \mathbb{N}^3$, which is an initially empty set that stores mappings of requests and processes from which they were received. It is primarily used to forward responses back to the processes that sent or relayed requests.

Let $configuration.participants$ denote the participant set of $configuration$ and let $configuration.round$ denote the round in which $configuration$ is used. Aside from the transitions determined by the MPTC instances a participant may be running, it additionally performs the following transitions:

- T1: **Precondition:** Received message $\langle \text{REQUEST}, pid', pid, (cid, rsn, cmd) \rangle$ where pid' is either a client or another participant, $rstate = \text{FALSE}$, $pid \notin configuration.participants$ and $(pid', cid, rsn) \notin responses$
- Action:** Add (pid', cid, rsn) in $responses$ and send $\langle \text{REQUEST}, pid, p, (cid, rsn, cmd) \rangle$ to each $p \in configuration.participants$.
- T2: **Precondition:** Received message $\langle \text{REQUEST}, pid', pid, (cid, rsn, cmd) \rangle$ where pid' is either a client or another participant, $rstate = \text{FALSE}$, $pid \in configuration.participants$, $\nexists (cid, rsn, *) \in requests$ and $(pid', cid, rsn) \notin responses$
- Action:** Add (cid, rsn, cmd) in $requests$ and (pid', cid, rsn) in $responses$. Send $\langle \text{REQUEST}, pid, p, (cid, rsn, cmd) \rangle$ to each $p \in configuration.participants \setminus \{pid\}$. If $|instances| < W$ then create an MPTC instance with instance id, $next-instance$ and proposal (cid, rsn) . Then add $(next-instance, cid, rsn)$ to $instances$ and update $next-instance = next-instance + 1$. Finally, run round $configuration.round$ of the new MPTC instance.
- T3: **Precondition:** Received message $\langle \text{REQUEST}, pid', pid, (cid, rsn, cmd) \rangle$ where pid' is either a client or another participant, $rstate = \text{FALSE}$, $pid \in configuration.participants$, $\exists (cid, rsn, *) \in requests$ and $(pid', cid, rsn) \notin responses$
- Action:** Add (pid', cid, rsn) in $responses$.

- T4: **Precondition:** Received message $\langle \text{REQUEST}, pid', pid, (cid, rsn, cmd) \rangle$ where pid' is either a client or another participant, and $rstate = \text{TRUE}$
- Action:** If $\nexists (cid, rsn, *) \in requests$ add (cid, rsn, cmd) in $requests$. If $(pid', cid, rsn) \notin responses$ add (pid', cid, rsn) in $responses$.
- T5: **Precondition:** Received message $\langle \text{RESPONSE}, rid, pid, (cid, rsn, result) \rangle$ from any replica or participant rid and $\exists (*, cid, rsn) \in responses$
- Action:** For each $(pid', cid, rsn) \in responses$, send $\langle \text{RESPONSE}, pid, pid', (cid, rsn, result) \rangle$ and remove (pid', cid, rsn) from $responses$.
- T6: **Precondition:** On round completion of some instance $(instance, cid, rsn) \in instances$ with outcome $(D, (cid, rsn))$ and $(cid, rsn, cmd) \in requests$
- Action:** Send $\langle \text{DECISION}, pid, rid, (instance, cid, rsn, cmd, configuration) \rangle$ to each $rid \in \mathcal{R}$. Then remove $(instance, cid, rsn)$ from $instances$ and (cid, rsn, cmd) from $requests$.
- T7: **Precondition:** $|instances| = 0$, $\exists (cid, rsn, cmd) \in requests$, $rstate = \text{FALSE}$ and $pid \in configuration.participants$
- Action:** Create an MPTC instance with instance id, $next\text{-}instance$ and proposal (cid, rsn) . Then add $(next\text{-}instance, cid, rsn)$ to $instances$ and update $next\text{-}instance = next\text{-}instance + 1$. Finally, run round $configuration.round$ of the new MPTC instance.
- T8: **Precondition:** On round completion of some instance $(instance, cid, rsn) \in instances$ with outcome $(M, (cid, rsn))$ or $(U, (cid, rsn))$ and there are still instances that have not completed $configuration.round$
- Action:** Update $rstate = \text{TRUE}$.
- T9: **Precondition:** On round completion of some instance $(instance, cid, rsn) \in instances$ with outcome $(M, (cid, rsn))$ or $(U, (cid, rsn))$, next configuration cfg and there no more instances that have not completed $configuration.round$

Action: Update $rstate = \text{FALSE}$ and $configuration = cfg$. Then send $\langle \text{RECONFIGURATION}, pid, p, (instances, requests, configuration, instance) \rangle$ to each $p \in configuration$. Finally, update $instances = \emptyset$ and $requests = \emptyset$.

T10: **Precondition:** Received messages $\langle \text{RECONFIGURATION}, p, pid, (instances_p, requests_p, configuration', instance_p) \rangle$ from a set of participants Q , $|Q| = f + 1$ such that $configuration'.round > configuration.round$

Action: Update $configuration = configuration'$, $next-instance = \max(instance_p \text{ for } p \in Q)$ and $requests = requests \cup (\cup_{p \in Q} requests_p)$. For each $p \in Q$ and $(cid, rsn, cmd) \in requests_p$ add (p, cid, rsn) in $responses$. For each $(instance', cid, rsn) \in \cup_{p \in Q} instances_p$ use Phase 3 of MPTC to update the proposal of $instance'$ and create an MPTC instance $(instance', cid', rsn')$ where (cid', rsn') is the updated proposal. Finally, add $(instance', cid', rsn')$ to $instances$.

Like in Turtle Consensus, MPTC is lazily instantiated for each slot. MPTC messages carry instance identifiers so incoming protocol messages are properly processed by the correct instance. If an instance has not yet been created, messages for that instance are queued and processed when it is created. Transition T2 ensures that if a request creates an instance to one of the active participants then unless the receiving participant fails, the remaining correct processes in the active participant set will also receive and create an instance for that request. Transition T7 ensures that if a proposed request does not get decided in one of the running instances, a new instance will be created for that proposal.

Replicas

The sets of replicas and participants are disjoint and $|\mathcal{R}| > f$ so at least one replica is always correct. Each replica in \mathcal{R} is run by a different process and maintains a copy of state of the service implemented by the SMRP. Replicas are responsible for executing the commands issued through clients' requests in the order established by the participants. This order is determined by the instance id's associated with each decision received. As in Section 5.4 we model this ordering as a sequence of numbered slots with the first slot numbered as 0. All replicas are initialized in the same state and since all commands are deterministic, executing commands in order at all replicas ensures that they all end up in the same state. To provide this functionality, replicas maintain the following state additional to the one related with the service SMRP implements:

- $rid \in \mathcal{R}$, initialized with the identifier of the replica.
- $execution-slot \in \mathbb{N}$, which is initialized 0 and maintains the position in the ordering of commands that should be executed next.
- $decisions \subseteq \mathbb{N}^3 \times Ops \times \mathcal{C}$, which is an initially empty set of mappings between a slot and a request as well as the configuration under which the request was decided.
- $completed \subseteq \mathbb{N}^3$, which is an initially empty set that stores mappings between slots and ids of requests that were executed in those slots.

In addition, to those transitions defined by the deterministic state machine implemented by the SMRP, replicas perform the following transitions:

- T1: Precondition:** Received message $\langle \text{DECISION}, pid, rid, (instance, cid, rsn, cmd, C) \rangle$ where $C \in \mathcal{C}$, $\nexists(instance, *, *, *, *) \in decisions$, and $execution-slot \leq instance$
- Action:** Add $(instance, cid, rsn, cmd, C)$ in $decisions$.
- T2: Precondition:** $\exists(instance, cid, rsn, cmd, C) \in decisions$, and $execution-slot = instance$, and $\nexists(*, cid, rsn) \in completed$
- Action:** Execute cmd and let $result$ be the outcome of the operation. Send $\langle \text{RESPONSE}, rid, pid, (cid, rsn, result) \rangle$ to each pid in the participant set specified by C . Add $(instance, cid, rsn)$ to $completed$ and remove $(instance, cid, rsn, cmd, C)$ from $decisions$. Update $execution-slot = execution-slot + 1$.
- T3: Precondition:** $\exists(instance, cid, rsn, cmd, C) \in decisions$, and $execution-slot = instance$, and $\exists(*, cid, rsn) \in completed$
- Action:** Add $(instance, cid, rsn)$ to $completed$ and remove $(instance, cid, rsn, cmd, C)$ from $decisions$. Update $execution-slot = execution-slot + 1$.

In other words, replicas execute commands one slot at a time as soon as they become available. Decisions already received or executed are ignored or treated as no-op. Note that there is still the issue of garbage collecting executed requests in *completed* that are no longer needed. In fact, *completed* can get arbitrarily large the longer the system operates. One solution would be to only track for each client the largest request identifier used such that future decisions about an already executed request number for a given client can be ignored.

6.6 Evaluation

In this section we present an evaluation of MPTC using the SMRP protocol presented in Section 6.5.1. In Section 6.6.1 we present the experiment setup and in Section 6.6.2 the performance results of MPTC under different attack scenarios.

6.6.1 Setup

We implemented MPTC and the SMRP described in Section 6.5.1 using C++. Our testbed consists of 10 nodes in Emulab [110], each with 8 cores running at 2.4 GHz, with 64GB of memory. For our experiments we used $f = 1$. Two nodes were designated as replicas, six as participants, one as clients, and one as the attacker. Nodes are connected by 1Gbps switched Ethernet as shown in Figure 6.2. Note that clients and attacker can only connect to participants, while participants connect to both replicas and clients. This choice was made to disable the attacker from directly attacking the replicas of SMRP and thus degrading performance without attacking the consensus mechanism. All communication between participants takes place through Switch 1 in our topology. Switch 2 is only used for participant to replica communication. We do not allow participants to communicate through Switch 2 since this would prevent the attacker from saturating the participants' bandwidth with respect to the MPTC execution. This would give MPTC an unfair advantage and would not showcase the benefits of its reconfiguration capabilities. All communication is over TCP/IP except for the DoS attack traffic, which is entirely UDP/IP. One of the two client nodes is used by the attacker and the other for creating legitimate client threads. As in Section 5.5 we use a separate node for

attacks in order to limit the effect of bandwidth attacks on the clients' ability to issue requests.

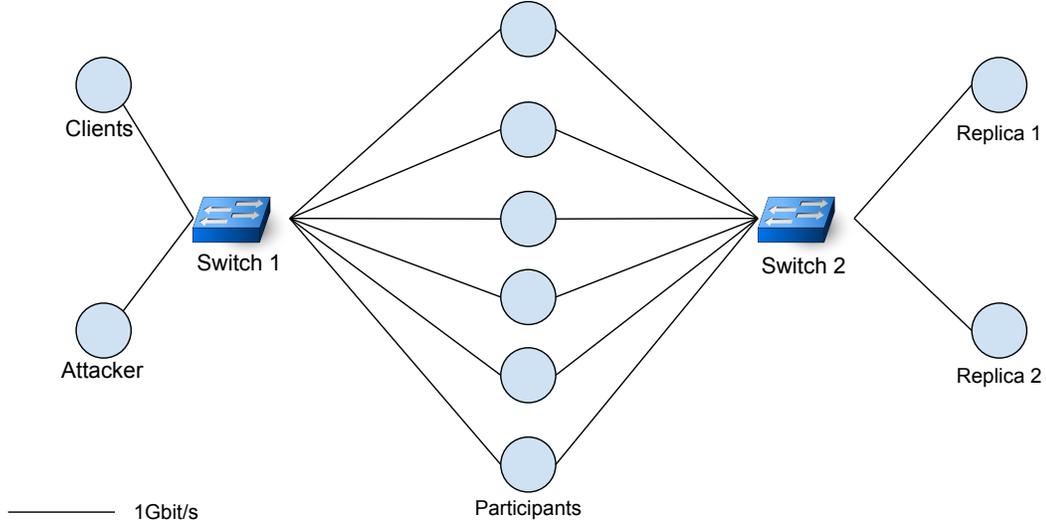


Figure 6.2: Experiment topology.

To simplify our evaluation, we set \mathcal{C} to contain only two configurations such that the corresponding participant sets are disjoint. The configuration selection function provided by the trusted dealer (in our implementation by a configuration file) simply alternates between these two configurations every time a round fails. Note that the leader of each configuration depends on the round in which the configuration is run as described in Section 6.5. We consider that the attacker does not have this knowledge to make informed decisions regarding targeting processes.

Clients first connect to $f + 1$ random participants to which they issues requests. Once connected, each client executes the following loop: It issues each request to all $f + 1$ participants, waits for a response, discarding duplicate responses, and then sends the next request. Note that by connecting to $f + 1$ participants, we ensure that each client request reaches at least one correct participant who will further forward the request to the active participants. We have client requests contain

no-ops, which means that when a decided request becomes ready for execution, replicas can immediately reply with a response.

The attacker creates a small number of attack threads, each of which targets a single participant, selects a random port, and sends UDP dummy messages as fast as it can. Note that these messages are not requests and are not processed by our participants since they never get to the application level. As in our Turtle Consensus evaluation, the goal of the attack is to prevent at most one participant from participating in MPTC instances. The attacker can focus all threads on the same participant or spread them across different ones. Since all attack threads are created on a single node, the aggregate bandwidth the attacker threads can saturate from the service cannot exceed 1Gbps.

We conducted experiments to test the throughput and latency of our implementation under normal execution and DoS attacks. Both metrics were measured at the client side. For throughput we measured the aggregate number of operations per second completed by client threads. Note that this is not the actual number of instances completed per second by our SMRP implementation since the same request might be decided more than once.

Other parameters of our experiment include:

- Duration: Each experiment lasted 1 minute. We found longer experiments did not significantly affect our metrics.
- Load: The number of concurrent clients, which ranged in our experiments from 1 to 64.
- Request size: The size of the command contained in each client request, which we set to 100 bytes.

- Attack message size: The size of the UDP messages send by attack threads to saturate the participants bandwidth; we set that to 1KB since our experimentation with our platform showed it is the smallest message size with the best results for the attacker.
- Number of attacker threads: Each run involving a DoS attack had 8 attack threads. We found that this number of threads yields best results for the attacker even when all 64 clients are connected to the target sharing the same link.
- Timeout: This is the initial timeout period used in our Paxos variant (Section 6.5) for each MPTC instance. Every time a round of some instance fails we double the timeout period for that instance.

6.6.2 Results

In our evaluation, we investigated three main scenarios. In the first, we run our implementation of MPTC without any attacks taking place. The performance of this scenario will be our baseline since any attack scenarios drain resources from the system and thus is expected to perform similar or worse. This scenario is labeled “No attacks” in our figures.

The second scenario has the attacker focusing the DoS attack on a single node, the one that hosts the Paxos leader. This attack depletes the leader’s bandwidth. In this scenario no reconfiguration occurs. More specifically, we assume that in each round of MPTC the exact same configuration is chosen and the leader remains the same. Note that this scenario tries to simulate the case where the adversary can accurately track and attack the leader of the Paxos configuration. While

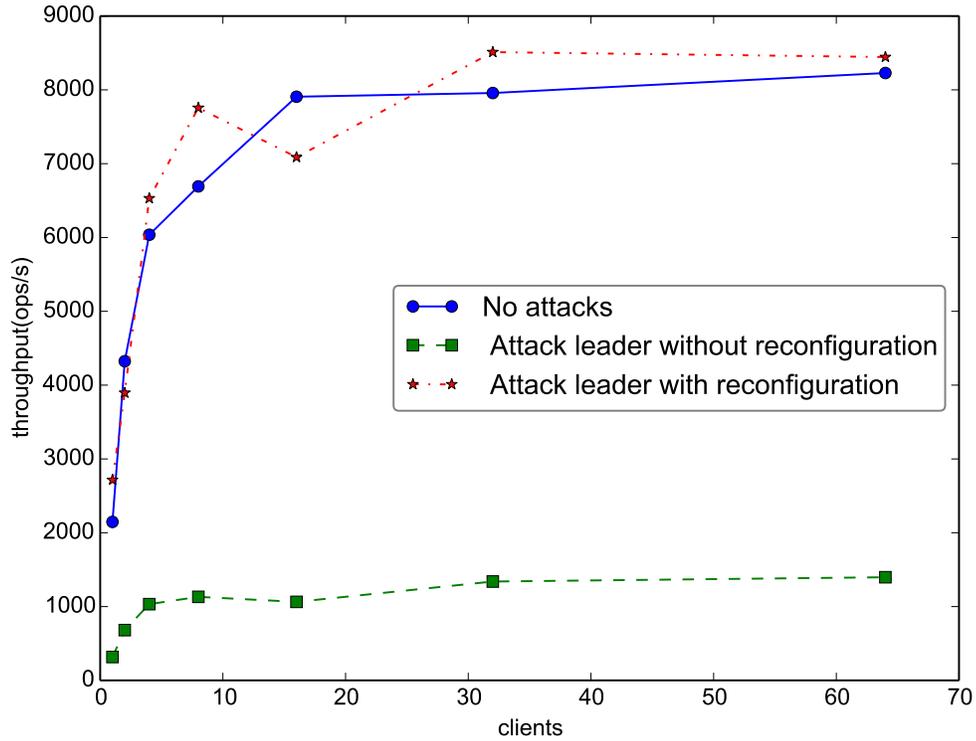


Figure 6.3: Throughput as a function of load and various attack scenarios.

any reasonable implementation of Paxos would change leaders among the $2f + 1$ processes, we set up the scenario to simplify issuing a very efficient attack. In our figures, this scenario is labeled “Attack leader without reconfiguration”.

Finally, the third scenario uses an attacker who like in the previous scenario focuses on a single node. In this scenario the attacker is given the initial position of the leader but this time our implementation uses the MPTC version we described in Section 6.6.1 where consensus instances execution alternates between two disjoint sets of nodes. The attacker strategy here is to saturate the bandwidth of the known leader. It keeps attacking that node for the entirety of the experiment run. This attack is labeled “Attack leader with reconfiguration”.

Figure 6.3 shows the throughput comparison of the previous three experiment scenarios as a function of the load on the SMRP. Each point represents the average throughput over 10 runs for each number of clients. In each of these runs clients connect to random participants, which in turn means that performance will vary across experiments. The first scenario is our best case scenario since the system operates at full resource capacity. The second scenario shows that performance suffers substantially when the Paxos leader is under attack. This is to be expected since the leader's participation is critical for making progress in each MPTC instance. In the third scenario we observe the benefits of the reconfigurable version of MPTC in action. The SMRP throughput is close to that of the No Attacks case. The main reason for this behavior is that since the leader of the first configuration is under attack and lacks the bandwidth to handle the valid traffic, some instance will inevitably fail the first round since the remaining participants will eventually time out. That will cause a reconfiguration that changes the active participant set. The new participants will pick up the failed instances as well as future requests and continue operating at full capacity. The minor deviations observed between scenarios 1 and 3 are mainly due to the randomness of client distribution over the set of all participants.

Figure 6.4 shows a comparison of the same scenarios as the load increases, but this time with respect to latency. Observe that all scenarios behave similarly with the latency linearly increasing with the load. This behavior is to be expected since, as the load increases, the number of concurrent MPTC instances increases, which in turn increases the latency for each client. After all, each of them has to wait for a response to their previous request before sending the next one. As in the case of throughput, we see that both scenarios 1 and 3 have similar latencies while scenario 2 performs poorly. The reasoning is again the same. In the second

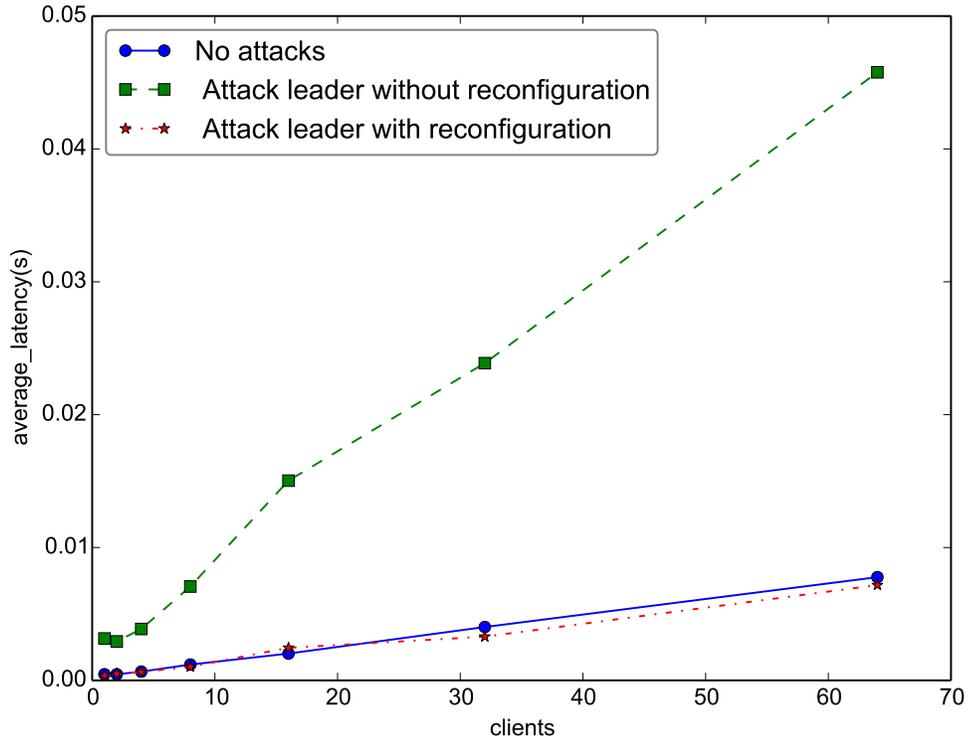


Figure 6.4: Latency as a function of load and various attack scenarios.

scenario the leader under attack is slower in completing instances, which raises the wait time for each client.

6.7 Concluding Remarks

In this chapter we presented Moving Participants Turtle Consensus (MPTC), an extension to the Turtle Consensus protocol presented in Chapter 5 that allows running different consensus protocols, on different sets of processes, across different rounds of a single consensus instance. By altering the execution configuration of the consensus mechanism on the fly we believe we can better protect a system under DoS attacks issued by adversaries that target configuration-specific vulner-

abilities. We described MPTC for both crash and byzantine failure environments. MPTC can deal with adversaries with bounded information on the system by making unpredictable changes in the execution of the protocol. Thus, our protocol eliminates adversary's advantage on predicting protocol and execution-specific information that can be used against it.

We built a prototype implementation of MPTC in which we used the same protocol across configurations and kept changing the set of processes executing the different rounds. Our evaluation suggested that we can achieve the performance offered by the most efficient consensus protocols even when the system is under attack. Thus, MPTC improves on the core Turtle Consensus protocol not only because it can handle stronger and better informed adversaries but also due to its ability to maintain good performance without having to resolve to less efficient consensus strategies.

There are various directions for further exploration and improvement of MPTC. First, MPTC should be tested under more sophisticated attacks. In our evaluation scenarios the attacker targeted the same node throughout the experiment so, once the configuration changes, the attacker can no longer affect the system. In a realistic environment that is not the case. Even without the configuration knowledge, the attacker can still pick another node at random when it detects that its attack does not substantially change the performance of the system. As the number of participants increases, however, the probability of a successful attack decreases. We leave more sophisticated implementations of the attacker for future work.

While our design focuses on attack tolerance, we believe that MPTC can be used in different environments where unexpected changes in the workload of the system may lead to sub-optimal configurations and thus reactive reconfiguration is

useful or even necessary. Such cases would benefit, however, from a more strategic and dynamic selection of configurations, which the current MPTC design lacks since it uses a predetermined sequence of configurations. We believe that finding efficient ways of dynamically selecting and applying new configurations is key for extending MPTC's flexibility and applicability.

CHAPTER 7

CONCLUSIONS

In this dissertation we presented two different approaches for designing distributed systems that can adapt to dynamic workloads. First, we studied proactive approaches in the context of cooperative caching for Online Social Networks. The suggested framework considers a service that acts as a directory for content locations and maintains information on social links between the clients of the service. The service provides the clients with hints about the location of requested content as well as where such content should be cached. We proposed two proactive cache placement approaches that use these social links between clients to suggest placement closer to the clients that are more likely to access it. One of these approaches allows for tweaking the degree of proactivity which controls how aggressively cache copies are created and thus the trade-offs between hit ratio and bandwidth cost. The evaluation and comparison of these approaches to commonly used reactive schemes was done via simulations on synthetically generated graphs and workloads whose characteristics match those of real OSNs. Our experiments show that substantial improvements can be achieved at moderate overheads using proactive strategies even under moderate churn, provided that appropriate workload information exists. In addition, comparison with optimal strategies that assume a-priori knowledge of the entire workload showed that proactive approaches can approximate optimal ones well.

There are many directions for expanding the work on proactive cooperative caching. One we consider to be of particular interest has to do with our parametrized cache placement approach, which used the replication factor parameter to control the number of copies made for a requested object. This parameter was

statically set during the beginning of each experiment. We believe that a reasonable extension would reactively update the value of this parameter during the operation of the system according to the hit ratio observed by the service to match the characteristics of the given setting and workload. Thus, such an extension could extend the applicability of our scheme to different content delivery settings.

Second, we presented a reactive moving target defense approach for dealing with certain Denial-of-Service attacks against fault tolerance mechanisms based on crash-tolerant consensus. Our protocol, that we call Turtle Consensus, operates by changing its execution configuration on-the-fly when no decision can be made under the current configuration. Reconfiguration happens in two dimensions. The first is the consensus strategy itself. This allows Turtle Consensus to deal with DoS attacks that are tailored for specific protocols. The second dimension under reconfiguration is the set of processes executing the protocol. In this extended version of Turtle Consensus, both the execution strategy and the participating processes can be updated in an unpredictable but predetermined fashion through cryptographic techniques. This eliminates the advantage of an adversary on predicting protocol execution details even under partial compromise of the system. Both the crash-tolerant and the byzantine tolerant versions of the Turtle Consensus protocol are presented, though our implementation and evaluation only considered crash failures. We tested the effectiveness of our approach with respect to both reconfiguration components. Using a prototype implementation that alternated between two dissimilar underlying consensus protocols, Paxos and Ben-Or, we found that we can achieve excellent performance under benign scenarios and acceptable performance while under attack, even under high client load. By additionally exploiting the changing participants capability of our design, we showed experimentally that we can further improve the behavior of Turtle Consensus and

maintain the performance offered by the most efficient consensus protocols even when the system is under attack.

There are various directions for further exploration and improvement of Turtle Consensus. One of the more immediate goals could be to test Turtle Consensus behavior under more sophisticated attacks as well as under a byzantine failure environment. As long-term goals, we consider that enhancing Turtle Consensus to dynamically select a configuration in an efficient manner could substantially extend its applicability in other dynamic settings.

Our study shows the importance of both proactive and reactive reconfiguration capabilities in distributed systems. We focused on concrete settings and carefully evaluated our techniques. We leave open to future research how the techniques can be applied to other distributed systems, as well as how both reactive and proactive approaches can be combined in the same system.

BIBLIOGRAPHY

- [1] <http://snap.stanford.edu/data/>. [Online; accessed 2012].
- [2] http://www.w3schools.com/html/html5_webstorage.asp. [Online; accessed 2014].
- [3] Finger Bomb. http://www.iss.net/security_center/reference/vulntemp/Finger_Bomb.htm, 1999. Accessed: 2016-07-18.
- [4] Marcos Kawazoe Aguilera and Sam Toueg. Randomization and failure detection: A hybrid approach to solve consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, WDAG '96, pages 29–39, London, UK, UK, 1996. Springer-Verlag.
- [5] Kemal Akkaya and Mohamed Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3(3):325 – 349, 2005.
- [6] Jamal N. Al-Karaki and Ahmed E. Kamal. Routing techniques in wireless sensor networks: a survey. *IEEE Wireless Communications*, 11(6):6–28, Dec 2004.
- [7] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, December 2004.
- [8] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: scaling file servers via cooperative caching. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 129–142, Berkeley, CA, USA, 2005. USENIX Association.
- [9] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441 – 461, 1990.
- [10] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

- [11] Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. DOS-resistant authentication with client puzzles. In *Security Protocols*, volume 2133 of *Lecture Notes in Computer Science*, pages 170–177. Springer Berlin Heidelberg, 2001.
- [12] Jean-Paul Bahsoun, Rachid Guerraoui, and Ali Shoker. Making bft protocols really adaptive. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 904–913, May 2015.
- [13] Laszlo A. Belady. A study of replacement algorithms for virtual storage. *IBM System Journal*, 1966.
- [14] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems*, 28(5):755 – 768, 2012. Special Section: Energy efficiency in large-scale distributed systems.
- [15] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 27–30, New York, NY, USA, 1983. ACM.
- [16] Fabrício Benevenuto, Tiago Rodrigues, Meeyoung Cha, and Virgílio Almeida. Characterizing user behavior in online social networks. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 49–62, New York, NY, USA, 2009. ACM.
- [17] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management*, MDM '01, pages 3–14, London, UK, UK, 2001. Springer-Verlag.
- [18] Sem Borst, Varun Gupta, and Anwar Walid. Distributed caching algorithms for content distribution networks. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, 2010.
- [19] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, October 1985.
- [20] Mike Burrows. The Chubby Lock Service for loosely-coupled distributed systems. In *7th Symposium on Operating System Design and Implementation*, OSDI'06, Seattle, WA, November 2006.

- [21] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.
- [22] 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.
- [23] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1), June 2006.
- [24] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.
- [25] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.
- [26] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.
- [27] Ilwoo Chang, Matti A. Hiltunen, and Richard D. Schlichting. Affordable fault tolerance through adaptation. In *Parallel and Distributed Processing, Lecture Notes in Computer Science 1388*, pages 585–603. Springer, 1998.
- [28] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.
- [29] Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM J. Comput.*, 23(4):701–712, August 1994.
- [30] Cisco. 7xx Router Password Buffer Overflow. <http://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-19971216-pw-buffer>, June 1998. Accessed: 2016-07-18.
- [31] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Sys-*

tems Design and Implementation, NSDI'09, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

- [32] Maria Couceiro, Paolo Romano, and Luis Rodrigues. A machine learning approach to performance prediction of total order broadcast protocols. In *Proceedings of the 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO'10)*, Budapest, Hungary, September 2010.
- [33] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A Hybrid Quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.
- [34] Diane Davidowicz. Domain Name System (DNS) Security. <http://compsec101.antibozo.net/papers/dnssec/dnssec.html>, 1999. Accessed: 2016-07-18.
- [35] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
- [36] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, January 1987.
- [37] Christos Douligeris and Aikaterini Mitrokotsa. DDoS attacks and defense mechanisms: classification and state-of-the-art. *Computer Networks*, 44(5):643 – 666, 2004.
- [38] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.
- [39] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.
- [40] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 427–438, Oct 1987.
- [41] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Proceedings of the 1983 International Conference on*

Fundamentals of Computation Theory, pages 127–140, London, UK, UK, 1983. Springer-Verlag.

- [42] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.
- [43] Haoyan Geng and Robbert van Renesse. Sprinkler—reliable broadcast for geographically dispersed datacenters. In David Eyers and Karsten Schwan, editors, *Middleware 2013*, volume 8275 of *Lecture Notes in Computer Science*, pages 247–266. Springer Berlin Heidelberg, 2013.
- [44] Xianjun Geng and Andrew B. Whinston. Defeating distributed denial of service attacks. *IT Professional*, 2(4):36–42, Jul 2000.
- [45] Jack Goldberg, Ira Greenberg, and Thomas F. Lawrence. Adaptive fault tolerance. In *Advances in Parallel and Distributed Systems, 1993., Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 127–132, Oct 1993.
- [46] Li Gong and Jack Goldberg. Implementing adaptive fault-tolerant services for hybrid faults, 1994.
- [47] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 363–376, New York, NY, USA, 2010. ACM.
- [48] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 314–329, New York, NY, USA, 2003. ACM.
- [49] Zygmunt J. Haas. A new routing protocol for the reconfigurable wireless networks. In *Universal Personal Communications Record, 1997. Conference Record., 1997 IEEE 6th International Conference on*, volume 2, pages 562–566 vol.2, Oct 1997.
- [50] Ligang He, Stephen A. Jarvis, Daniel P. Spooner, and Graham R. Nudd. Optimising static workload allocation in multiclusters. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 39–, April 2004.

- [51] Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 175–188, Berkeley, CA, USA, 2008. USENIX Association.
- [52] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of Facebook photo caching. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'13)*, 2013.
- [53] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.
- [54] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the Twenty-First Annual Symposium on Principles of distributed computing*, PODC '02, pages 213–222, New York, NY, USA, 2002. ACM.
- [55] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: Transparent moving target defense using software defined networking. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 127–132, New York, NY, USA, 2012. ACM.
- [56] Sushil Jajodia, Anup K Ghosh, Vipin Swarup, Cliff Wang, and X Sean Wang. *Moving target defense: creating asymmetric uncertainty for cyber threats*, volume 54. Springer Science & Business Media, 2011.
- [57] Flavio P. Junqueira, Patrick Hunt, Mahadev Konar, and Benjamin Reed. The ZooKeeper Coordination Service (poster). In *Symposium on Operating Systems Principles (SOSP)*, 2009.
- [58] Muhammad Kafil and Ishfaq Ahmad. Optimal task assignment in heterogeneous distributed computing systems. *IEEE Concurrency*, 6(3):42–50, Jul 1998.
- [59] Malachi Kenney. Ping of Death. <http://insecure.org/splotts/ping-o-death.html>, January 1997. Accessed: 2016-07-18.

- [60] Sherif. M. Khattab, Chatree Sangpachatanaruk, Rami Melhem, Daniel Mosse, and Taieb Znati. Proactive server roaming for mitigating denial-of-service attacks. In *Information Technology: Research and Education, 2003. Proceedings. ITRE2003. International Conference on*, pages 286–290, Aug 2003.
- [61] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07*, pages 45–58, New York, NY, USA, 2007. ACM.
- [62] John Kubiawicz, 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(11):190–201, November 2000.
- [63] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
- [64] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.
- [65] Felix Lau, Stuart H. Rubin, Michael H. Smith, and Ljiljana Trajkovic. Distributed denial of service attacks. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 3, pages 2275–2280 vol.3, 2000.
- [66] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, KDD '05*, pages 177–187, New York, NY, USA, 2005. ACM.
- [67] Bo Li, Mordecai J. Golin, Giuseppe F. Italiano, Xin Deng, and Kazem Sohraby. On the optimal placement of web proxies in the Internet. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1282–1290 vol.3, 1999.
- [68] Xiaoming Liu, Robbert van Renesse, Mark Bickford, Christoph Kreitz, and Robert Constable. Protocol switching: Exploiting meta properties. In *International Workshop on Applied Reliable Group Communication at the Inter-*

national Conference on Distributed Computing Systems (ICDCS), Phoenix, AZ, April 2001.

- [69] Thanasis Loukopoulos and Ishfaq Ahmad. Static and adaptive data replication algorithms for fast information access in large distributed systems. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, pages 385–392, 2000.
- [70] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys Tutorials*, 7(2):72–93, Second 2005.
- [71] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Symposium on Operating System Design and Implementation (OSDI)*, pages 105–120. USENIX, December 2004.
- [72] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08*, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.
- [73] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies, FAST '03*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.
- [74] Ravi Mirchandaney, Don F. Towsley, and John A. Stankovic. Adaptive load sharing in heterogeneous systems. In *Distributed Computing Systems, 1989., 9th International Conference on*, pages 298–306, Jun 1989.
- [75] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, April 2004.
- [76] Amir-Hamed Mohsenian-Rad and Alberto Leon-Garcia. Distributed internet-based load altering attacks against smart power grids. *IEEE Transactions on Smart Grid*, 2(4):667–674, Dec 2011.
- [77] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13*, pages 358–372, New York, NY, USA, 2013. ACM.

- [78] Mark E. J. Newman, Steven H. Strogatz, and Duncan J. Watts. Random graphs with arbitrary degree distributions and their applications. *Phys. Rev. E*, 64:026118, Jul 2001.
- [79] Stavros Nikolaou and Robbert van Renesse. Turtle consensus: Moving target defense for consensus. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 185–196, New York, NY, USA, 2015. ACM.
- [80] Stavros Nikolaou, Robbert van Renesse, and Nicolas Schiper. Cooperative client caching strategies for social and web applications. In *Large-Scale Distributed Systems and Middleware (LADIS)*, Farmington, PA, November 2013.
- [81] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.
- [82] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO '91*, pages 129–140, London, UK, UK, 1992. Springer-Verlag.
- [83] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Survey of network-based defense mechanisms countering the dos and ddos problems. *ACM Comput. Surv.*, 39(1), April 2007.
- [84] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 43–57, Oakland, CA, May 2015. USENIX Association.
- [85] Lili Qiu, Venkata N. Padmanabhan, and Geoffrey M. Voelker. On the placement of web server replicas. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1587–1596. IEEE, 2001.
- [86] Michael O. Rabin. Randomized byzantine generals. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 403–409, Nov 1983.
- [87] Venugopalan Ramasubramanian, Zygmunt J. Haas, and Emin Gün Sirer. Sharp: A hybrid adaptive routing protocol for mobile ad hoc networks. In *Proceedings of the 4th ACM International Symposium on Mobile Ad Hoc*

Networking & Computing, MobiHoc '03, pages 303–314, New York, NY, USA, 2003. ACM.

- [88] Lakshmith Ramaswamy, Ling Liu, and Arun Iyengar. Cache clouds: Co-operative caching of dynamic documents in edge networks. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS 2005)*, pages 229–238, 2005.
- [89] Kavitha Ranganathan and Ian T. Foster. Identifying dynamic replication strategies for a high-performance data grid. In *Proceedings of the Second International Workshop on Grid Computing*, GRID '01, pages 75–86, London, UK, 2001. Springer-Verlag.
- [90] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.
- [91] Robert Constable, Mark Bickford, Robbert van Renesse. Investigating correct-by-construction attack-tolerant systems. Technical report, Cornell University, 2010.
- [92] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.
- [93] Alessandra Sala, Lili Cao, Christo Wilson, Robert Zablit, Haitao Zheng, and Ben Y. Zhao. Measurement-calibrated graph models for social network experiments. In *Proceedings of the 19th International Conference on World wide web*, WWW '10, pages 861–870, New York, NY, USA, 2010. ACM.
- [94] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.
- [95] Livia M. R. Sampaio, Francisco V. Brasileiro, Walfredo Cirne, and Jorge C. A. Figueiredo. How bad are wrong suspicions? towards adaptive distributed protocols. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pages 551–560, June 2003.

- [96] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.
- [97] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.
- [98] Victor Shoup. Practical threshold signatures. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT’00, pages 207–220, Berlin, Heidelberg, 2000. Springer-Verlag.
- [99] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. BFT protocols under fire. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI’08, pages 189–204, Berkeley, CA, USA, 2008. USENIX Association.
- [100] Tyron Stading, Petros Maniatis, and Mary Baker. Peer-to-peer caching schemes to address flash crowds. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS ’01, pages 203–213, London, UK, 2002. Springer-Verlag.
- [101] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM ’01, pages 149–160, New York, NY, USA, 2001. ACM.
- [102] Asser N. Tantawi and Don Towsley. Optimal static load balancing in distributed computer systems. *J. ACM*, 32(2):445–465, April 1985.
- [103] Duc A. Tran, Khanh Nguyen, and Cuong Pham. S-clone: Socially-aware data replication for social networks. *Computer Networks*, 56(7):2001 – 2013, 2012.
- [104] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using ensemble. *Softw. Pract. Exper.*, 28(9):963–979, July 1998.
- [105] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in Horus. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC ’95, pages 80–89, New York, NY, USA, 1995. ACM.

- [106] Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using Ensemble. *Software–Practice and Experience*, 28(9), August 1998.
- [107] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffei. Horus: A flexible group communication system. *Commun. ACM*, 39(4):76–83, April 1996.
- [108] Robbert van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, May 2003.
- [109] Jia Wang. A survey of web caching schemes for the Internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, October 1999.
- [110] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI’02)*, pages 255–270, Boston, MA, December 2002. Usenix.
- [111] Alec Wolman, Geoffrey M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M. Levy. On the scale and performance of cooperative web proxy caching. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, SOSP ’99*, pages 16–31, New York, NY, USA, 1999. ACM.
- [112] Anthony D. Wood and John A. Stankovic. Denial of service in sensor networks. *Computer*, 35(10):54–62, Oct 2002.
- [113] Liang Zhang, Fangfei Zhou, Alan Mislove, and Ravi Sundaram. Maygh: building a CDN from client web browsers. In *Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys ’13*, pages 281–294, New York, NY, USA, 2013. ACM.
- [114] Lidong Zhou. Towards fault-tolerant and secure on-line services. Technical report, Cornell University, Ithaca, NY, USA, 2001.