System Support for Efficient Replication in Distributed Systems

Thèse N° 9548

Présentée le 6 septembre 2019

à la Faculté informatique et communications
Laboratoire de calcul distribué
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Dragos-Adrian SEREDINSCHI

Acceptée sur proposition du jury

Prof. M. Odersky, président du jury
Prof. R. Guerraoui, directeur de thèse
Prof. F. Pedone, rapporteur
Prof. M. K. Reiter, rapporteur
Prof. E. Bugnion, rapporteur

2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

To my family and friends,
as well as my id, ego, and super-ego.

"there is no bull!"
— Sheldon B. Kopp

Acknowledgements

My family was outstanding in their patience and encouragement. They helped me keep my feet firmly planted, always remembering the important parts of life.

I am grateful to my advisor Prof. Rachid Guerraoui for his patience, guidance, and relentless confidence throughout these (almost) six years. Beyond his research craftsmanship, I often see in Rachid a glimpse of what I imagine to be the mien of a Zen master—the outlook of a composed person always vibrating on his very particular positive note. So I learned from him some of the secrets to good research, but also how to lead an all-round well-lived life.

As part of my thesis defense, I am indebted to Prof. Martin Odersky, who marshaled the discussion flawlessly as the president, as well as to Prof. Edouard Bugnion, who never fails to bring the profoundest of discussion points in every meeting he joins. Prof. Fernando Pedone and Prof. Mike Reiter joined as external examiners, both of whom I respect immensely. Their question, comments, and suggestions helped me get a clear understanding of the limitations and possible extensions of the work in this dissertation.

I spent the summer of 2016 as a research intern with Dr. Marko Vukolić at IBM Research in Zurich. I am thankful for those great four months and the skilled questions Marko always brought in discussion—in fact, I am still thinking about some of those questions.

The summer of 2017 is among the most unforgettable periods of my studies. I worked under the guidance of Dr. Dahlia Malkhi at the VMware Research Group in Palo Alto, California. Prior to that, I only knew Dahlia by name and by an implicit respect, owing to her foundational work on the field in which I work today. Since then, my respect for her only grew. Some of the people I met in Palo Alto make me wishfully indulge in ideas of moving to the (great) America.

Several additional professors and collaborators have been instrumental in my development. I appreciate the kind mentorship of Prof. Anne-Marie Kermarrec, Prof. Babak Falsafi, Prof. George Candea, Prof. Willy Zwaenepoel, Prof. Katerina Argyraki, Dr. Yvonne-Anne Pignolet, Dr. Jad Hamza, Dr. Diego Didona, and Prof. Petr Kuznetsov.

Beside professors and researchers, I crossed paths with numerous other bright and amazing people. It would be unfair to mention any name here, since I would certainly though unwittingly miss many. But such is life—unfair—so I will mention at least some of my closest friends. Julia, Giel, and Victor are the first people I met in Lausanne. They always strive (and succeed) to keep a balance in their life, unlike me, but I made great progress of late. They are decisively the most friendly and cheerful among my friends, and I am grateful for all of the delightful moments we passed together.

My lab is great. I joined with the notorious cohort of 2013, alongside George, Matej, Mahsa,

Jingjing, and Rhicheek. David, Tudor, and Vasilis steered us in the proper direction and helped us develop the same high standards in our research as their own. I still rely on their advice, and regard them as essential parts of my life in Switzerland. They also helped immensely to level-up my standards when it comes to the venues where I hang out. Along the way, Igor, Karolos, and George (the younger) also joined, and have proved to be shaping forces in my social and professional life. More recently, I am happy to have also greeted in the lab Matteo, Thanasis, and Arsany. I am thankful to everyone for the numerous enlightening discussions. It has been great to share so many exceptional moments (technical discussion, "ups and downs," parties, trips, etc.) with you. I am looking forward to our surf workshops.
Kristine Verhamme, Fabien Salvi, and notably France Faille have always been friendly and helpful with administrative (and not only) challenges. Past and current flatmates, lifting buddies, and friends from around EPFL have been a relentless—and much needed—source of joy and inspiration. People are great.
My work has been supported in part by the European ERC Grant 339539 - AOC.

Lausanne, 9th of May 2019 **D.-A. SEREDINSCHI**

Preface

This thesis presents part of the research towards a Ph.D. under the supervision of Prof. Rachid Guerraoui in the School of Computer and Communication Science at EPFL in Switzerland, between 2013 and 2019. The main results in the thesis appear originally in the following publications (author names appear in alphabetical order).

1. Rachid Guerraoui, Jad Hamza, Dragos-Adrian Seredinschi, and Marko Vukolić. "On the Non-Scalability of State Machine Replication". Under submission. (Part II)
2. Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. "Incremental Consistency Guarantees for Replicated Objects". In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16). (Part III)
3. Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. "Exo: System Support for Efficient Token Transfers". Under submission. (Part IV)

In addition to the results presented in this thesis, the following conference publications contain additional work (author names appear in alphabetical order).

1. Rachid Guerraoui, Anne-Marie Kermarrec, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. "Atum: Scalable Group Communication Using Volatile Groups". In Proceedings of the 17th International Middleware Conference (Middleware 2016).
2. Alain Girault, Gregor Gössler, Rachid Guerraoui, Jad Hamza, and Dragos-Adrian Seredinschi. 2018. "Monotonic Prefix Consistency in Distributed Systems". In Proceedings of the 38th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2018).
3. Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. 2018. "State Machine Replication Is More Expensive Than Consensus". In Proceedings of the 32nd International Symposium on Distributed Computing (DISC 2018).
4. Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. "Harmony: a scalable and decentralized trust infrastructure" . In Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN 2019).

5. Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. “The Consensus Number of a Cryptocurrency”. In Proceedings of the 38th Annual ACM Symposium on Principles of Distributed Computing (PODC 2019).

Abstract

Current online applications, such as search engines, social networks, or file sharing services, execute across a distributed network of machines. They provide non-stop services to their users despite failures in the underlying network. To achieve such a high level of reliability, these applications rely on a simple technique: replication. Briefly, there are copies of the application on multiple machines, such that no specific machine represents a single point of failure. Under this apparent simplicity, however, reliability comes at a steep price. This is because replication entails certain side-effects (e.g., overheads or costs, inconsistencies, or tradeoffs) which often lead to inefficient designs and want for better performance.

The high-level problem we study in this dissertation is that of obtaining good performance in replicated systems. We seek to *reduce*—and when irreducible, *hide*—the effects of replication. We approach this problem from three fronts, as follows.

First, we look at State Machine Replication (SMR). This is a classic technique for achieving strong consistency in replicated systems. Similar to other techniques for strong consistency, SMR brings a substantial performance overhead. Additionally, this overhead gets worse with growing system size: There is a performance decay as an SMR system gets larger. We shed light on this issue by deploying five SMR systems on up to 100 replicas and reporting on their performance decay. Towards mitigating this issue, we introduce Carousel, an SMR system based on a ring overlay network which can alleviate performance decay.

Second, we discuss a technique for hiding the cost of strong consistency. Given the high overhead of accessing data with strong consistency, we propose that applications combine multiple consistency models. We introduce programming support for doing so, through an abstraction called Correctables. In conjunction with a speculation-based technique, we show how Correctables can lower the latency for strongly-consistent operations.

Third, we propose a technique to bypass SMR (and avoid its costs) in a concrete replicated application. We focus on the problem of implementing token transfer applications (e.g., online payments). We introduce the abstraction of exclusive token accounts, or Exa, supporting asynchronous transfers. We also design and build Astro, a system implementing the Exa abstraction. This system departs from classic consensus-based solutions (i.e., SMR), and relies instead on a broadcast primitive, a more efficient and tractable building block.

Keywords: replication, abstractions, consistency, efficiency, distributed storage, state machine replication, speculation, consensus, broadcast, token transfer.

Résumé

Les applications en ligne modernes, comme les moteurs de recherche, les réseaux sociaux ou les services de partage des données, utilisent un réseau distribué de machines. Ils assurent un service sans interruption pour leur utilisateurs malgré la présence de failles dans le réseau. Pour fournir un tel niveau de fiabilité, ces applications comptent sur une technique assez simple : la réplication. En bref, il y a des copies de l'application sur plusieurs machines, de sorte qu'aucune machine ne soit un point critique du réseau. Malgré cette simplicité apparente, néanmoins, la fiabilité représente un énorme coût. La réplication des données engendre des problèmes (coûts additionnels, incohérences, ou compromis à trouver), dont la résolution repose souvent sur des méthodes peu efficaces.

De façon général, la problème abordé dans cette thèse est la conception de systèmes répartis fiables et efficaces. Notre but c'est de *réduire*—ou, si n'est pas possible, de *masquer*—les conséquences indésirables de la réplication. Nous traiterons ce problème sur trois niveaux.

Premièrement, nous considérons la réplication des états des machines (State Machine Replication, SMR), une technique classique pour la copie de données avec un haut niveau de cohérence (strong consistency). Comme d'autres techniques similaires, SMR implique une réduction considérable des performances. Pire, les performance sont d'autant plus détériorée que la taille du système est grande : la performance de la SMR décroît au fur et à mesure que système croît. Dans cette thèse, nous déploierons cinq systèmes SMR sur jusqu'à cent copies pour analyser le déclin de la performance. En vue de rectifier ce problème, nous introduirons Carousel, un système SMR qui s'appuie sur un réseau avec une topologie en anneau.

Deuxièmement, nous présenterons une technique pour cacher les coûts de la réplication. Étant donné les coûts élevés d'accès aux données répliquées avec une garantie d'un haut niveau de cohérence, nous proposerons l'utilisation des plusieurs niveaux de cohérence par les applications réparties. Nous introduirons le support de programmation pour accomplir cela, avec une abstraction appelée Correctables. À l'aide d'une technique par spéculation, nous montrerons comment Correctables peut diminuer les délais d'accès aux données répliquées.

Troisièmement, nous proposerons une technique pour contourner l'utilisation de SMR (et de ses hauts coûts) dans une application répartie. Nous nous arrêterons sur le problème d'implémenter les applications fondées sur des *tokens* (comme le paiement en ligne). Nous introduirons une abstraction des comptes exclusivement dédiés à l'utilisation de tokens, appelée Exa,

qui permet des transferts asynchrones des tokens entre les utilisateurs. Nous proposerons aussi la conception du système Astro, qui implémente cette abstraction Exa. Ce système est different de ses prédécesseurs qui s'appuient sur le consensus (i.e., SMR), parce qu'il utilise une technique de diffusion de messages plus efficace et plus simple.

Mots clés : réplication, abstraction, consistance, efficacité, systèmes répartis de données, réplication des machines aux états, spéculation, consensus, diffusion, transferts des tokens.

Contents

Acknowledgements **v**

Preface **vii**

Abstract (English/Français) **ix**

List of Figures **xiv**

List of Code Listings **xvi**

List of Tables **xvii**

Introduction **1**
Thesis Context 2
Thesis Roadmap 2

I Preliminaries **5**

1 Background **7**
1.1 Replication in Distributed Systems 7
1.2 Techniques for Efficient Replication 9

2 Overview **13**
2.1 Thesis Statement 13
2.2 Thesis Contributions 13

II On the Performance Decay of State Machine Replication **15**

3 Observing the Performance Decay of State Machine Replication **17**
3.1 Introduction 17
3.2 Background 20
3.3 Methodology 22
3.4 Empirical Study 25

4 Carousel: Mitigating Throughput Decay in State Machine Replication **33**
4.1 Carousel Common-Case Protocol 33
4.2 Carousel Reconfiguration 38

5 Discussion & Concluding Remarks **41**
5.1 Discussion 41
5.2 Conclusions 44

III Supporting Efficient Access to Replicated Data **45**

6 Incremental Consistency Guarantees **47**
6.1 Introduction 47
6.2 Overview & Related Work 50
6.3 Correctables 53
6.4 Correctables in Action 55
6.5 Bindings 61
6.6 Evaluation 63
6.7 Concluding Remarks 72

IV Supporting Efficient Token Transfers **73**

7 Asynchronous Token Transfers **75**
7.1 Introduction 75
7.2 Overview 77
7.3 Astro System 79
7.4 Experimental Evaluation 86
7.5 Discussion & Related Work 96
7.6 Concluding Remarks 99

Conclusions **101**

A Carousel Supplementary Material **103**
A.1 Safety and Liveness Guarantees for Carousel 103

B Astro Supplementary Material **105**
B.1 Safety and Liveness Guarantees for Astro 105
B.2 Byzantine FIFO Broadcast Algorithm 107

Bibliography **124**

Curriculum Vitae **125**

List of Figures

3.1 Throughput decay for five SMR systems on a public wide-area cloud platform. 26
3.2 Average latencies for experiments with five SMR systems. 29
3.3 Stability experiments for ZooKeeper, etcd, and for BFT-Smart. 30

4.1 Overview of Carousel SMR system. 34
4.2 The unfolding of an Append operation in Carousel. 36

6.1 In the tradeoff between consistency and performance, many applications fall into a *gray zone*, torn between the need for both. 48
6.2 High-level view of Correctables, as an interface to the underlying storage. . . . 53
6.3 The three states, transitions, and callbacks associated with a Correctable object. 54
6.4 Server support for efficient ICG. 62
6.5 Single-request latencies in Cassandra for different quorum configurations. . . . 65
6.6 Performance of Correctable Cassandra compared to baseline Cassandra. 66
6.7 Divergence of preliminary from final (correct) views in Correctable Cassandra with various YCSB configurations. 66
6.8 Efficiency of the ICG implementation in Correctable Cassandra (CC). 67
6.9 Latency gap between preliminary and final views on experiments with Correctable ZooKeeper compared to baseline ZooKeeper. 68
6.11 Using speculation via ICG to improve latency in an advertising system and in Twissandra. 70
6.12 Improving the latency in a ticket-selling application. 71

7.1 Dependencies among transfer operations in a token-based system. 78
7.2 Throughput evolution with a growing system size, considering two token-transfer systems, one based on broadcast, and another based on consensus (i.e., SMR). Regional scenario. 89
7.3 Throughput evolution with a growing system size, considering two token-transfer systems, one based on broadcast, and another based on consensus (i.e., SMR). Global scenario. 90
7.4 Throughput-latency graph for $N = 10$, comparing Astro against a consensus-based solution. The deployment environment is global. 91
7.5 Throughput-latency graph for $N = 100$, comparing Astro against a consensus-based solution. The deployment environment is global. 91

7.6 Throughput-latency graph for balanced versus single setups in Astro. We use $N = 100$ replicas in the global environment. 92
7.7 Throughput evolution when we introduce a crash-stop failure of a single replica in the Consensus-based system (either the leader or a random replica) and the Astro broadcast-based system (a random replica). 93
7.8 Throughput evolution in the consensus- and broadcast-based systems when we introduce asynchrony ($100ms$ delay for each outgoing packet) at a replica, either the leader or a random one. 94
7.9 Throughput evolution for $N = 100$ when a crash-stop failure or asynchrony affects the consensus-based system or the broadcast-based system. 95

List of Code Listings

4.1 A high-level algorithm describing the Append operation in Carousel. 35
6.1 Different consistency guarantees in Reddit. 56
6.2 Reddit code rewritten using Correctables. 56
6.3 Generic speculation with Correctables. 57
6.4 Example of applying speculation in an advertising system to hide latency of strong consistency. 57
6.5 Dynamic selection of consistency guarantees in a ticket selling system. 59
6.6 Progressive display of news items using Correctables. 59
6.7 Simple binding to a storage system with primary-backup replication. 62
7.1 The interface of an Exa object. 77
7.2 The state at each node in Astro. 81
7.3 Algorithm for the transfer operation in Astro. 81
7.4 Byzantine FIFO broadcast callback in Astro. 82
7.5 A node in Astro applies a transfer operation on its local state. 82
7.6 A node delivers a proof message for a transfer. 83
7.7 Validity checks for a transfer message in Astro. 84
7.8 Verification of dependencies and their certificate. 84
7.9 Computing the balance of an Exa object in Astro. 85
B.1 Pseudocode for Byzantine FIFO broadcast. 108

List of Tables

3.1 Round-trip latencies between different regions in the SoftLayer cloud. 23

6.1 Different types of applications building on top of replicated objects. 52

Introduction

The Master said:
One who would study for three years
Without thought of reward
Would be hard indeed to find

— Confucius, translated by A. Waley [Wal00, Book VIII]

Distributed systems are the backbone of modern society. Whether we are doing groceries, enrolling in a college, or bypassing traffic jams on our way to work, most of our activities somewhere along the line involve a distributed system. This technology both empowers and simplifies our life. Relying on this technology, however, can also burden us in uniquely modern ways. For instance, a source of anxiety in modern life has to do with the possibility of losing our precious data—financial, medical, or legal documents—or even worse, being robbed of our privacy—relating to confidential records such as photographs or conversations.

One of the main merits of distributed systems is their reliability. This high-level property simply states that a system as a whole has no single point of failure, and can withstand a wide array of faults. Even if computers constantly crash and churn, our data survives. Even if attackers gain control over a node in a distributed system, they can inflict limited damage.

Replication is the basic technique behind reliable distributed systems. Under a replicated design, the algorithms controlling our data, as well as the data itself, have multiple copies. Each copy is called a replica and resides in a different location. Often, the replicas of a system are spread across different regions of the globe. A replicated system can thus overcome single points of failure via redundancy. A serious problem such as a natural disaster may affect a replica and wipe out the corresponding data at a specific location; nevertheless, another replica in the system can assume the responsibilities of the affected replica.

Replication, however, does not come for free. Under a replicated design, the basic challenge is to keep each replica in the system consistent with the others. A replication protocol achieves this task, often called a coordination protocol. Informally, such a protocol seeks to maintain an identical state across all replicas, regardless of the application workload which constantly modifies this state, regardless of machine or network failures, or other such interference. From

the perspective of a user, replication protocols seek to effectively hide the existence of multiple copies—masking inconsistencies, failures, delays—and provide the illusion of a single, reliable machine. If the replication protocol achieves this, we say that the protocol ensures strong consistency guarantees.

In the absence of a replication protocol to ensure strong consistency, users may observe anomalous application behavior. For instance, an e-mail client might display different versions of an inbox, depending on which replica that client connects to. One of the replicas might maintain the up-to-date version of inbox, with all the recently received e-mails, while another replica maintains a stale version of this inbox. Ideally, the client should always display the most fresh version of the inbox, by reading from the up-to-date replica. But what if the connection to this replica is extremely slow—then should the client display the correct inbox (i.e., achieve strong consistency) despite unusual delays? Or would it be more expedient to provide a fast response and display the stale (i.e., inconsistent) inbox?

Thesis Context

The e-mail application we highlight above is a prime example of a central dilemma in replicated systems: Choosing between a consistent (but slow) and an inconsistent (but usually faster) response. Indeed, seminal results in the research literature establish that there are inherent costs associated with replication [AW94, LSP82, Lam83, MA06], and that some coordination problems are even impossible to solve under certain conditions [BT85, Bre12, FLP85, GL02]. Generally speaking, these results mark the existence of a tradeoff between consistency and performance in replicated systems, as described in the e-mail example.

At a high-level, throughout this thesis we study closely this consistency versus performance tradeoff. Our goal is to make replication protocols more efficient, in an attempt to lower the user-perceived cost of replication. To do so, we study both replication protocols (seeking to understand their limitations and boost their performance), as well as the application-level requirements (seeking to exploit application semantics and offer support for specific tasks).

Thesis Roadmap

In broad strokes, this dissertation spans across four parts, as follows.

Part I—Preliminaries

In this first part of the dissertation we cover essential background work on replication protocols (Chapter 1). We also describe earlier approaches towards achieving efficient replication. We conclude this part with a brief thesis statement and our contributions (Chapter 2).

Part II—On the Performance Decay of State Machine Replication

In the second part of this dissertation we focus on a classic approach for building reliable distributed systems, called State Machine Replication (SMR). This is a general-purpose solution for achieving strong consistency in a broad range of applications. Owing to its generality, SMR is among the most intensely studied approaches to replication.

In the context of SMR, we study how the degree of replication (i.e., system size) impacts performance. More precisely, we evaluate five SMR systems, including state-of-the-art implementations such as ZooKeeper or etcd. SMR systems such as these typically execute at small size, e.g., using three replicas. The assumption is that their performance decays sharply as their size increases. There is not much experimental evidence of this decay, however.

We deploy the SMR systems on up to 100 replicas and find that they exhibit performance decay with increasing replication degree, confirming prior observations at smaller scale. We notice, however, that we can partially mitigate this decay. Concretely, our study includes two research prototypes meant to explore the limits of SMR performance, namely ChainR and Carousel. These two employ efficient dissemination overlays (based on chain and ring overlays, respectively) which can curb performance decay. (Chapter 3). Our hope is that these results motivate further research on the performance decay of SMR.

In this part we also present the rationale behind different design choices in the Carousel SMR system (Chapter 4), and then conclude with a discussion (Chapter 5).

Part III—Supporting Efficient Access to Replicated Data

In the third part of the dissertation we propose a solution for tackling the tension between consistency and performance. Given the inherent costs of each individual consistency model (i.e., the overheads of strong consistency, or the anomalies of weak consistency), we propose that applications combine multiple consistency models. For supporting this task, we introduce the Correctables abstraction. The central feature of this abstraction is that it provides efficient access to multiple consistency levels for every operation. Correctables hide the implementation details for achieving different levels of consistency and the resulting complexity, allowing programmers to focus on balancing consistency with performance in their applications.

We exploit an observation that appears repeatedly in practical workloads, namely that that weakly-consistent versions of data tend to be often correct (i.e., consistent) in fact. Using the Correctables abstraction, we show how an application can speculate on weakly-consistent data towards hiding the latency of strong consistency (Chapter 6).

Part IV—Supporting Efficient Token Transfers

In the fourth part of the dissertation we turn our attention to token-based applications (e.g., implementing online payments). These applications typically build on SMR, and consequently inherit the overheads associated with this technique, including performance decay (as studied in Part II). We propose to bypass SMR (and avoid its costs) by describing the exclusive token account—or Exa—abstraction. This abstraction supports efficient implementations of token transfers. Concretely, Exa allows to eschew the need for imposing a total order across all transfers in the system (i.e., using SMR). This translates into an implementation that is simpler and more efficient than SMR-based solutions.

The Exa abstraction models a container for tokens (e.g., a wallet). The defining characteristic of an Exa object is that it has a single owner, and this owner is the only one allowed to spend money (i.e., initiate transfers) from that object. Owing to this exclusive-ownership property, Exa permits weak ordering of transfers, avoiding consensus. We design and build Astro, a system implementing the Exa abstraction. Compared to a baseline based on SMR, Astro is simpler, more robust to asynchrony or faults, and offers better performance (Chapter 7).

Conclusions and Appendix

In the concluding chapter of this dissertation we give a brief review of our contributions, discuss limitations of our techniques, as well as future work (Conclusions).

This dissertation also contains an appendix. We provide supplementary details for the Carousel system (Appendix A) as well as for the Astro system (Appendix B).

Preliminaries Part I

1 Background

My intention is to confuse you with only one thing at a time.

— Joe Armstrong [Arm13]

In this chapter we cover background notions on reliable distributed systems, with a focus on replication. This chapter spans two sections, as follows. First, we discuss basic concepts, presenting concrete approaches to replication as well as their applications (§1.1). Second, we look at replicated systems from a performance standpoint, describing techniques for improving their efficiency (§1.2).

1.1 Replication in Distributed Systems

To mask failures, reliable distributed systems rely on the idea of quorums. These are subsets of system replicas which have the property that any two such subsets intersect in at least one correct replica [Gif79]. A correct replica is one which does not crash (nor is compromised by an adversary, depending on the system model). A quorum typically comprises a fraction of the whole system. Even if some replicas fail, it is assumed that sufficiently many replicas remain alive (and non-compromised) such that a quorum of them is responsive.

Intuitively, the intersection property of quorums helps ensure safety, while their responsive nature is important for liveness [AS85, Lam77]. By safety we understand that the responses we obtain from a replicated system are in some sense correct (for example, an e-mail client should not display outdated appointments) and by liveness we mean that this system eventually responds (the e-mail client is guaranteed to connect). This abstraction of quorums is essential to providing fault-tolerance, and underlies most of the approaches and techniques we consider in this dissertation. Depending on the system model and assumptions, there are different flavors of quorum systems, providing different safety and liveness properties [BVF+12, MR97].

State Machine Replication. One of the most popular approaches to building reliable distributed systems is State Machine Replication [Sch90]. With SMR, the replicas of a system first agree on the order of incoming operations (e.g., client requests), and then execute these operations in the chosen order. For reaching agreement on the order of operations, the replicas in SMR employ a consensus algorithm.

There are various algorithms for implementing consensus, but at a high-level they all achieve the same goal, namely: output a single decision (i.e., the next operation to execute) among multiple concurrent proposals [CGR07, HKJR10, Lam03]. As we explained earlier, to mask faulty nodes consensus algorithms also rely on an underlying construction of quorum systems.

Being a general-purpose technique, SMR lends itself to a broad range of applications. For instance, this approach found adoption in various cluster systems, for implementing distributed lock services [Bur06] or other high-level abstractions [MMN+04]. In the context of wide-area systems, SMR can be used to build distributed key-value storage systems [GBKA11] or other applications [MJM08].

SMR ensures strong consistency, or linearizability [HW90]. As we mentioned earlier, this means that SMR hides from the application any side-effects of replication (such as faults or inconsistencies). This approach is able to withstand both simple crash failures, and even arbitrary, i.e., Byzantine, faults [LSP82]. As the topic of Byzantine failures represents a central concern in this thesis, we go into more details below.

Byzantine Fault-Tolerance. In the Byzantine fault model, a fraction of replicas may suffer from arbitrary faults. For instance, a faulty replica may even exhibit adversarial behavior towards the rest of the system. This behavior may arise as a consequence of software bugs, network interference, or malicious participants. Byzantine fault-tolerant (BFT) replication is an approach that seeks to secure against this broad class of faults.

There are reasons to believe that BFT systems impose larger overheads compared to crash fault-tolerant (CFT) systems [LVC+16]. Consequently, there is a long line of work on making BFT replication more efficient. The goal is to mitigate the performance penalties of BFT replication and make this technique more appealing [BSA14, Cas01, GAG+19, KAD+07, PP13]. Notwithstanding these efforts and their success, the practical need for BFT replication has been repeatedly debated in the past [BCvR09, CFJS12, CWA+09, HKJR10, WKR+13].

A relatively recent use-case for BFT replication is in the implementation of decentralized applications where participants are mutually distrustful of each other. Informally, these are called digital trust applications. Examples are cryptocurrencies [Nak08], secure storage [Woo15], or decentralized computing [HMW]. The rise in popularity of these applications can be mostly attributed to the success of the Bitcoin cryptocurrency (i.e., token transfer system) [Nak08].

Nakamoto's Bitcoin protocol, introduced in a 2008 white paper, implements a token transfer system without any central authority [Nak08]. Since then, many alternatives to Bitcoin came to prominence, e.g., Ethereum [Woo15], ByzCoin [KJG+16], Hyperledger Fabric [ABB+18], Tendermint [Ten19], or HotStuff [YMR+18]. Each of these alternatives brings novel approaches to implementing decentralized transfers, and some go beyond transfers, offering a more general interface in the form of smart contracts [Sza97]. Regardless of their novelty, most solutions in the sphere of digital trust applications employ consensus-based algorithms.

Despite a growing body of literature for implementing various types of digital trust applications, token transfer remains a central research topic and one of the most tenable representative of a digital trust system [Cou19, Mal18]. In Part IV of this dissertation we focus on this very type of application, presenting an abstraction that supports efficient token transfers.

We note that systems implementing digital trust can be broken down in two categories, depending on their assumption. Some are designed for a permissioned model, i.e., a private environment where all participants know each other. Others are for a permissionless model, i.e., a public setting allowing open participation. Throughout this thesis, we are interested in systems for the permissioned setting.

Replicated Data Types. Many distributed applications build upon specific types of objects, e.g., locks, storage, or queues. For instance, commercial websites can model their shopping cart as an instance of a read-write register (i.e., key-value storage) [DHJ+07]. Another example is that of a ticket shop where unsold tickets can be modeled as a replicated queue. Upon buying a ticket, each customer dequeues an element from this replicated object.

As we argued earlier, SMR supports the building of arbitrary replicated services, including replicated data types such as registers or queues. Protocols for SMR, however, can be less efficient than protocols for replicating a specific object, hence the need for direct support of replicated data types. This is just one example of a technique for gaining efficiency in a replicated system, and we expand on this discussion in the following section.

1.2 Techniques for Efficient Replication

Building on a specific data type in a distributed application can be seen as a form of exploiting application semantics. This is a classic technique for extracting more performance in replicated systems. We now expand on the discussion of this technique, and then explore two others, specifically: sharding and tweaking the system model.

Exploiting Application Semantics. Adopting a replicated data type—instead of a general-purpose solution based on SMR—is desirable not only from the standpoint of performance, but also for bypassing some fundamental limitations of SMR. For instance, the FLP impos-

sibility result applies to SMR (or, equivalently, to any consensus algorithm) and states the following: Absent any synchrony assumption, it is not possible for nodes in a distributed system to reach consensus if even one node can fail [FLP85]. In other words, SMR implementations are impossible in certain systems, such as asynchronous networks. This stands in contrast with atomic read-write register algorithms, for instance, which are possible in such a model [ABND95].

Even if specific data types permit more efficient implementations than SMR, and even bypass certain impossibility results, algorithms for replicated objects experience their own tradeoffs. For instance, the CAP theorem applies to replicated storage. This result asserts that access to a replicated storage system can not guarantee both availability and strong consistency in the presence of network partitions [Bre12]. In light of this result, many applications opt for weak consistency models as an effective technique for the ability to guarantee high-availability [DHJ+07]. Not only that, but in exchange for weak consistency, better performance and simpler solutions are often possible [ABK+15, LFKA11].

Roughly speaking, the weaker an abstraction is, the more efficient an implementation tends to be possible in practice. Therefore, applications which simply do not require strong abstractions (such as SMR for achieving strong consistency) frequently build on weaker ones. The same principle—exploiting application semantics—is at play. An interesting example of this can be found in applications that exploit commutativity. More precisely, certain applications allow relaxing of coordination requirements, i.e., ordering of operations in the system, if their operations commute [BR92, CKZ+15, Her90]. CRDTs [SPBZ11] or RADTs [RJKL11] are examples of this technique in practice.

One important challenge that appears when adopting weak abstractions is that programming (i.e., application-level logic) can become more difficult. The CAP theorem and other similar formulations [Aba12, AW94, MAD11, GGG+18, TPK+13] bring into focus the various tradeoffs and the resulting complexity of programming with replicated objects when considering different consistency models. In Part III of this dissertation, we will broach this subject in more detail: We revisit the topic of efficient access to replicated objects, and present an abstraction that supports programming under different consistency models.

Sharding. To mask a certain number F of faulty replicas, SMR systems for the crash-fault asynchronous model typically require $N = 2F + 1$ total replicas. Every operation that enters the system executes on a quorum of at least $F + 1$ replicas before responding to the client. In a similar vein, BFT systems typically employ quorums of $2F + 1$ replicas in a system of $N = 3F + 1$ nodes. For instance, to tolerate 3 Byzantine faults, such a system requires 10 replicas, and each operation must be accepted by at least 7 replicas.

The bounds on system and quorum sizes spawn from the need to guarantee the quorum properties we mentioned earlier in the beginning of this chapter. This degree of replication applies not only to SMR, but to any algorithm seeking to ensure strong consistency. For

applications where the state is very large (e.g., distributed transactional databases), or where it is desirable to accommodate a large number of participants, this high degree of replication can pose a bottleneck. It is typically assumed that some replicated systems (in particular SMR) experience a sharp performance decay with increasing system size. Consequently, typical use-cases of SMR employ this technique using a minimal number of replicas, typically three or five nodes; indeed, in Part II of this dissertation we study the matter of performance decay (specifically for SMR) in more detail.

Sharding is a classic technique in distributed systems, allowing to parallelize the processing of client operations, and hence go beyond the full replication model. For instance, a sharded SMR solution can allow separate groups of replicas to handle operations on disjoint partitions of the application state [BPR14, CDG+08, GBKA11].

Tweaking the System Model. Beside sharding, another method that enables reliable distributed systems to grow to large scales without incurring significant performance decay is to switch to a system model with probabilistic guarantees. There is a growing body that does so, in the context of consensus or other replication algorithms. In exchange for an arbitrarily small probability of breaking certain properties, these solutions can accommodate a large number of participants. This technique of adopting probabilistic guarantees found adoption in many use-cases, e.g., in replicated storage systems [BVF+12, MRWW01], in protocols for cryptocurrencies [EGSVR16, GHM+17, KKJG+17, Nak08], as well as in group communication systems [GKPS16].

Other system model tweaks that can yield more efficient designs in reliable distributed systems include synchrony assumptions [Ten19, VRS04] or hardware-assisted solutions [ISAV16]. While these systems offer interesting design choices, throughout this dissertation we are interested in replicated systems that offer deterministic guarantees, and designed for commodity networks.

2 Overview

2.1 Thesis Statement

In this dissertation we study techniques for reducing the costs of replication in reliable distributed systems. These costs can take various forms, e.g., inconsistencies (if using certain weak consistency models), overheads in terms of latency or throughput or performance decay with increasing system size (if using strong consistency), etc. The challenge is to provide applications with strong consistency guarantees without incurring substantial penalties on their performance. We approach this challenge from both a low-level perspective (studying building blocks and primitives directly) and a high-level perspective (seeking to understand the precise requirements of applications and exploit their semantics).

2.2 Thesis Contributions

This thesis comprises three high-level contributions [GHSV19, GKM+18, GPS16].

1. We take steps to further our understanding of SMR performance decay due to increasing system sizes. As a concrete approach towards mitigating SMR performance decay, we introduce the Carousel SMR system.
2. We introduce Correctables, an abstraction supporting efficient access to replicated objects. This abstraction simplifies and enables low-latency operations on replicated data.
3. We present Exa, an abstraction supporting efficient transfer operations for token-based applications, by eschewing the need for a consensus (i.e., SMR) algorithm. We also design, implement, and evaluate Astro, a system implementing the Exa abstraction.

Part II

On the Performance Decay of State Machine Replication

3 Observing the Performance Decay of State Machine Replication

What is in the end to be shrunk
must first be stretched.

— Arthur Waley [Wal13, Ch. XXXVI]

In this chapter we seek to discover how the size—i.e., the degree of replication—of SMR systems affects their performance. We deploy five SMR systems on a wide-area network, and we report on how their performance decays with increasing system size for up to 100 replicas. Our focus is mostly on throughput, but we cover latency as well.

We draw two high-level observations. (1) We notice that, informally, SMR throughput decays as a function of system size. An interesting pattern appears, however: This decay is initially sharp (consistent with studies at smaller scale), but as each system grows beyond a few tens of replicas, the decay dampens. In other words, the decay rate is exponential. (2) We observe that SMR systems based on chain or ring overlays can escape the exponential decay rate: We can characterize their performance as decaying at a linear, slow rate.

3.1 Introduction

As we discussed earlier (§1.1), State Machine Replication (SMR) is a classic approach to building distributed systems that are reliable and guarantee strong consistency [Sch90]. While this approach lends itself to a broad range of applications, it is interesting to note than most deployments of SMR are in small system sizes, e.g., three or five replicas for the crash-tolerant case [ABB+18, CDE+13, Bur06, HKJR10].

According to distributed systems folklore, SMR is too expensive if deployed on more than a handful of replicas [BP16, CML+06, GKPS16, HKJR10]. Indeed, there is a tradeoff between performance and fault-tolerance, and some performance decay is inherent in SMR [AEMGG+05]. As an SMR system grows in size, it is capable to tolerate more faults—but performance does not grow accordingly. In fact, performance typically decays with increasing system size. This

is because the system replicas (or more precisely, a quorum of replicas) must agree on every operation. Hence, as the number of replicas increases, the cost for reaching agreement increases as well.

A few workarounds exist to deal with SMR performance decay. First, some systems employ SMR to a limited degree. They ensure strong consistency for only a small, critical subset of the system state (e.g., metadata or configuration), while the rest of the system has a scalable design under weaker guarantees [ADW10, GGL03, QAB+13]. The critical part of the system builds on mature SMR systems such as ZooKeeper [HKJR10], etcd [Etc19a, Ong14], Consul [MPS93], or Boxwood [MMN+04].

A second workaround is sharding [GBKA11, ALvRV13]. In this case, the service state is broken down into disjoint shards, each shard running as a separate SMR cluster [BP16, CDE+13]. Additional mechanism for cross-shard coordination, such as 2PC [BBC+11], ensures that the whole system behaves consistently. Yet a third workaround consists in abandoning strong consistency, so as to eschew SMR altogether [Bre12, Vog09].

Briefly, the purpose of these workarounds is to avoid executing SMR at a larger scale. Consequently, SMR systems have almost never been deployed, in practice, on more than a few replicas, typically three to five [CDE+13]. So today there is not much empirical evidence of SMR throughput decay as system size grows. For instance, we do not know how ZooKeeper [HKJR10] or PBFT [CL02] perform with, say, 100 replicas.

This question—of SMR performance decay in practice—is not only of academic interest. For example, SMR protocols are important in decentralized services, standing at the heart of digital trust applications (e.g., distributed ledgers) in permissioned environments [ABB+18, SBV18]. For these applications, SMR protocols are expected to run on at least a few *tens* of replicas [CDE+16, Vuk15]. Another example is in a sharded design: Under Byzantine fault-tolerant models, shards should not be too small, otherwise an adversary may easily compromise the system. In this case, it is critical that each shard comprises tens or hundreds of replicas [KKJG+17, GKPS16]. But SMR systems struggle from the “SMR does not scale” stigma and the lack of experiments with increasing system size.

In this chapter we address the void in the literature by deploying and observing how the size of an SMR system impacts its performance. Our primary goal is to obtain empirical evidence of how SMR performance decays in practice at larger size. We deploy and evaluate five SMR systems on up to 100 replicas and report on our results.

The first three systems we study are well-known SMR implementations: ZooKeeper [HKJR10] and etcd [Etc19a], which are crash fault-tolerant (CFT), and BFT-Smart [BSA14], which is Byzantine fault-tolerant (BFT). Consistently with previous observations [BP16, CML+06, HKJR10], we observe that their throughput decays sharply at small scale. The interesting part is that this trend of sharp decay does not persist. Throughput decay dampens as systems get larger, so overall their throughput follows an exponential decay rate.

ZooKeeper, etcd, and BFT-Smart execute most efficiently—obtain best performance—when deployed at their smallest size, i.e., running on 3 replicas (4 for BFT-Smart). Throughput drops to 50% of its best value at 11 replicas. When running on 50 replicas, the throughput decays to almost 10%. On 100 replicas the throughput drops to roughly 6% of its best value. In absolute numbers, these systems sustain 300 to 500 tps (transactions per second) at 100 replicas on modest hardware in a public cloud platform. The average latency is below 3.5s, while the 99th percentile is 6.5s, even for the BFT system.

These three systems are hardened SMR implementations and we choose them for their maturity. We complement the performance observations with a stability study. Briefly, we seek to understand whether these systems are capable to function despite faults at large scale. More precisely, we inject a fault in their primary (i.e., leader) replica and evaluate their ability to recover. We find that ZooKeeper recovers excellently (in a few seconds), indicating that this system can perform predictably at scale, for instance to implement a replicated system across hundreds of nodes. The other two systems are slower to recover or have difficulties doing so at 100 replicas.

The fourth system we investigate is ChainR, based on chain replication [VRS04]. This system is throughput-optimized, so it helps us delineate the ideal case, namely, a throughput upper bound. When growing from 3 to 100 replicas, throughput in ChainR decays very slowly, from 15k tps to 11k tps (i.e., to 73% of its best value). If we place replicas carefully on the wide-area network so as to minimize chain traversal time, ChainR exhibits below 3 seconds latency.

It can be misleading, however, to praise chain replication as the ideal SMR protocol. ChainR does not suffer from performance decay as severely as others, indeed—but only in graceful executions (i.e., failure-free and synchronous network). In non-graceful runs, throughput drops to zero. This protocol sacrifices availability, because it must reconfigure (pausing execution) to remove any faulty replica [VRS04]. This system relies on a synchronous model with fail-stop faults, a strong assumption. Worse still, a single straggler can drag down the performance of the system, since the chain is only as efficient as its weakest link.

The fifth SMR system we study employs a ring overlay, a generalization of chain overlays. We call this system Carousel, and we design it ourselves. In contrast to ChainR, this system does not pause execution for reconfiguration, maintaining availability despite asynchrony or faults.

Unlike prior solutions, Carousel does not rely on reconfiguration [VRS04, VRHS12] nor a classic broadcast mode [AGK+15, Kne12] for masking faults or asynchrony. Doing so would incur downtime and hurt performance. Instead, we take the following approach: Each replica keeps fallback (i.e., redundant) connections to other replicas. When faulty replicas prevent (or slow down) progress, a correct replica can activate its fallback path(s) to restore progress and maintain availability. The goal of this simple mechanism in Carousel is to selectively bypass faults or stragglers on the ring topology (preserving good throughput).

In a 106-replica system, Carousel sustains $6k$ tps when there are no faults, and throughput

decays to 55% of its best value. If $F = 21$ replicas manifest malicious behavior, then throughput reaches $4k$ tps (48% decay). Since Carousel and ChainR are research prototypes, we do not evaluate their stability, which we leave for future work.

To summarize, in this chapter we investigate how performance decays in SMR systems as we increase their size. We deploy five SMR systems using at least 100 replicas in a geo-replicated network. We observe that, indeed, throughput decays in these systems as a function of system size, but this decay dampens. Our experiments with chain- and ring-replication show that there are ways to mitigate throughput decay in SMR, informing future designs.

We organize the rest of this chapter as follows. We first provide more background on SMR, including the systems under our study (§3.2), and then discuss our methodology for evaluating these systems (§3.3). Next, we present our observation on the performance decay of five SMR systems (§3.4), and conclude (§5.2). We take a rather unconventional approach of presenting first our observations on Carousel (in §3.4), and only in the subsequent chapter discussing its design (Chapter 4).

3.2 Background

SMR systems often employ a modular design. In Paxos terminology, for instance, there is a distinction between proposers, acceptors, and learners (or observers) [VRSS15]. Proposers and acceptors handle the agreement protocol, while learners execute operations. Throughout this chapter we are interested in the agreement protocol.

While execution can dominate the latency and cost in SMR systems [YMV+03], this step is not subject to performance decay. The agreement protocol is the one that typically encounters inefficiencies as systems get larger. As we mentioned, certain applications require agreement to execute on tens or hundreds of replicas, e.g., for decentralized services, or to ensure that shards are resilient against a Byzantine adversary [CDE+16, KKJG+17, GKPS16, Vuk15].

When increasing the size of an SMR system, some performance degradation is unavoidable. This is inherent to replicated systems that ensure strong consistency, because a higher degree of replication (i.e., fault-tolerance) entails a bigger overhead to agree on each client operation. But how does performance decay—in a linear manner? Or does the decay worsen or does it lessen when system size increases?

Both throughput and latency are vital measures of performance, and it is well known that these two are at odds with each other in SMR systems [Lam03]. Our interest is on throughput decay, but we also cover latency results.

3.2.1 SMR Systems in Our Study

Our study covers five SMR protocols.

ZooKeeper. This system is based on ZAB, an atomic broadcast algorithm [JRS11], and is implemented in Java. We study ZooKeeper rather than ZAB directly, as ZAB is tightly integrated inside the ZooKeeper system.

etcd. This system is implemented in Go and is based on the Raft consensus algorithm [Ong14].

ZAB and Raft share many design points [Ong14], and given their similarities, we expect that ZooKeeper and etcd exhibit similar performance decay in practice. Both of these systems are actively maintained and widely used in production. They have found adoption in cluster and multi-datacenter (WAN) networks alike [Etc19c, AGTK15, BJS11, Fou12].

BFT-Smart. The third system we study is implemented in Java and provides BFT guarantees [BSA14]. BFT-Smart is actively used and has been maintained by a team of developers for over eight years, being a default choice for prototyping research algorithms in several groups [Bft19a, LVC+16, PLL+15]. This system is patterned after the seminal PBFT consensus algorithm of Castro and Liskov [CL02].

ZooKeeper, etcd, and BFT-Smart employ a *leader-centric* design [BMSS12, Ong14], i.e., they rely on the leader replica to carry most of the burden in the agreement protocol. Specifically, the leader does not only establish a total-order across operations, but also disseminates (via broadcast) those operations to all replicas. Typically, the network overlay has the shape of a star, with the leader in the center. This design simplifies the SMR algorithm [Ong14]. The disadvantage is that the leader replica (its CPU or bandwidth) can become the bottleneck [JRS11].

We choose these three systems for their maturity: These are production-ready (ZooKeeper and etcd) or seasoned implementations (BFT-Smart). We also study the stability of these three systems, i.e., executions where the leader replica crashes. Prototypes, like the next two systems we consider, may deliver better performance, but often do so without vital production-relevant features which, once implemented, can hamper performance.

The fourth and fifth SMR systems we wrote ourselves: ChainR, our prototype of chain replication [VRS04], and Carousel, a ring-based replication protocol with BFT guarantees. Both are written in Go. Chain replication, and in particular its ring-based variants, are provably throughput-optimal in uniform networks [GLPQ10, JMPSP17]. In contrast to leader-centric protocols, these systems avoid the bottleneck at the leader. This is because the network overlay is more efficient, balancing the burden of dissemination across system replicas.

ChainR. We use this system as a baseline, to obtain an ideal upper bound—and what other SMR systems could aim for—in terms of both absolute throughput and throughput decay. We faithfully implement the common-case with pipelining and batching [Ong14, SS12, VRS04].

ChainR works in a fail-stop model, i.e., assumes synchrony to mask crash faults [VRS04], unlike the other four systems we study. Mechanisms to make chain- or ring-based systems fault-tolerant include an external reconfiguration module, or a special recovery mode [AGK+15, ALvRV13, Kne12, VRHS12]. Such mechanisms degrade availability, as even simple crashes put

the system in a degraded mode, possibly for extended periods of time.

Carousel. This system represents our effort in designing a BFT protocol optimized for throughput, which can withstand sub-optimal conditions (e.g., faults, asynchrony) and hence offer improved availability. We briefly describe Carousel below, and additionally dedicate a full chapter for more details (Chapter 4).

Carousel is an asynchronous ring-based SMR system. When a fault occurs in Carousel, we mask this by temporarily increasing the fanout at a particular replica. This is in contrast to prior ring- or chain-based designs, which resort to a recovery mode or reconfiguration.

Every replica in Carousel has a default fanout of 1, i.e., it forwards everything it receives to its immediate successor on ring. In the worst case, F consecutive replicas on the ring can be faulty. In this case, the predecessor of all these nodes (a correct replica) increases its fanout to $F+1$, bypassing all F faults. This way, the successor of these faulty nodes still receives all updates propagating on the ring, and progress is not interrupted. The system throughput deteriorates when this happens, but not as badly as that of broadcast-based solutions, where one of the replicas—the leader—has a fanout of $2F+1$ (or $3F+1$ for BFT) [CL02, HKJR10].

3.3 Methodology

We now discuss the testbed for our study (§3.3.1), details of the workload (§3.3.2), as well as the workload suite we use to conduct experiments (§3.3.3).

3.3.1 Testbeds

We consider as testbed the SoftLayer public cloud platform spanning multiple datacenters [Sof]. We use virtual machines (VMs) equipped with 2 (virtual) CPU cores and 4GB RAM. We use low spec-ed VMs to gain insight in SMR performance on commodity hardware. In all our tests we place each client and replica in a separate VM. This separation avoids unnecessary noise in our results, which would happen if there was contention for local resources.

Network. The bandwidth available between different nodes, either clients or replicas, is set at 100Mbps. Latencies in SoftLayer range from under 10ms to almost 200ms, depending on distance. We consider nine regions of North, Central, and South America. We use `ping` to measure the inter-regional latencies (which are symmetric), and present our results in Table 3.1.

Node placement. As Table 3.1 illustrates, there is a large disparity in cross-regional latencies. Consequently, replica and client placement across regions can impact performance. By default, we always place all clients in Washington. Spreading clients randomly has no benefit, and would introduce unnecessary variability in the results. For ZooKeeper, etcd, and BFT-Smart, we place replicas randomly across the nine regions.

	MON	TOR	DAL	SEA	SJC	HOU	MEX	SAO
WDC	15	22	31	56	60	39	56	115
MON		9	38	61	64	43	63	123
TOR			30	53	56	37	55	124
DAL				40	36	8	25	143
SEA					18	48	65	174
SJC						44	56	195
HOU							30	136
MEX								167

Table 3.1 – Round-trip latencies (msec) between regions in SoftLayer. The regions are: Washington (WDC), Montreal (MON), Toronto (TOR), Dallas (DAL), Seattle (SEA), San Jose (SJC), Houston (HOU), Mexico (MEX), and Sao Paolo (SAO).

Replica placement is particularly important for chain- or ring-based systems. For instance, in ChainR, client requests propagate from one replica to another, starting from the head of the chain until it reaches the tail; the tail responds to clients. If we distribute replicas randomly, then requests will pass back and forth between regions, accumulating latencies on the order of seconds or worse. Random replica placement in ChainR is unreasonable. Instead, successive nodes in the chain should be clustered in the same region, and the jumps across regions should be minimized, i.e., the latency for traversing the chain should be minimal.

We take a simple approach to replica placement for ChainR and Carousel. We start with Washington, and then traverse the continent from East to West and North to South—i.e., counter-clockwise—as follows: (1) Washington, (2) Montreal, (3) Toronto, (4) Dallas, (5) Seattle, (6) San Jose, (7) Houston, (8) Mexico, and (9) Sao Paolo. Each region hosts a random number of replicas between 8 and 12. Note that placing clients in Washington does not give any advantage to ChainR and Carousel, since each client request has to traverse the whole network.

While this is not the optimal method to place replicas, it is simple and yields surprisingly good results (§3.4.1). More complex alternatives exist to our heuristic-based solution. Indeed, optimizing placement in geo-replicated settings is useful not only in cases such as ours, but also to minimize service costs or optimize other metrics [WBP+13].

Operating system and software. All machines in our study run Ubuntu 14.04.1x64. For Apache ZooKeeper, we use $v3.4.5$ [Zoo14]. We install etcd $v2.3.7$ directly from its repository [Etc19b]. For BFT-Smart we use $v1.2$ [Bft19b].

3.3.2 Workload Characteristics

The goal of our workload is to stress the central part of SMR systems, their agreement algorithm. In practice, this algorithm is also in charge of replicating the request payload (i.e., blocks of transactions) to all replicas. Clients send requests of 250 bytes, inspired from a Bitcoin workload which is typically considered when the peak throughput is discussed [CDE+16].

Requests are opaque values which replicas do not interpret. The execution step consists of simply storing the payload of each request in memory. We omit having an explicit execution step which is application-dependent. This step is often embarrassingly parallel, and optimizations at this level are orthogonal to our observations [ADP18, KWQ+12].

Local Handling of Requests. The component which handles requests differs slightly among the systems we consider. For instance, ZooKeeper and etcd ensure *persistent* storage of requests by writing them to a file system, whereas the other systems do not have a persistence layer. To ensure a fair comparison, we mount a `tmpfs` filesystem and configure ZooKeeper and etcd to write requests to this device. In BFT-Smart we handle requests via a callback, which appends every request to an in-memory linked list. ChainR and Carousel simply log each request to an in-memory array (i.e., Go slice).

Batching. We do not optimize the batching configuration. This is because different system sizes usually require different batch sizes to optimize throughput [MXC+16]. Moreover, batching is often an implementation detail hidden from users, e.g., etcd hardcodes the batch size to 1MB [Etc19d]. Similarly, batching in ZooKeeper is not configurable; this system processes requests individually, and batching seems to be handled entirely at the underlying network layer (Netty). In BFT-Smart we use batches of 400, the default. In ChainR and Carousel we are more conservative, allowing up to 10 requests per batch, since these systems are already throughput-optimized at their network overlay level.

It is well-known that batching affects the absolute throughput of a system. We are primarily interested, however, by the throughput *decay* function of SMR systems. We do not seek to maximize absolute throughput. Prior work has shown that batching does not affect the throughput decay, e.g., in BFT SMR systems [CML+06, §6]. In light of this, we expect that the throughput decay in each system evolves independently of batch size.

3.3.3 Workload Suite

Our workload suite has two parts. We use (1) a workload *generator* that creates requests and handles the communication between clients and each SMR service. We also use (2) a set of scripts to *coordinate* the workload across all clients and control the service-side (e.g., restart between subsequent experiments). The workload generator differs across SMR systems, since each system has a different API. The coordinating side is common to all systems.

The main components of the workload generator are a client-side library, which abstracts over the target system, and a thread pool, e.g., using the `multiprocessing.Pool` package in Python, or `java.util.concurrent.Executors` in Java. The pool comprises parallel workload generators in each client machine. For ZooKeeper, etcd, ChainR, and Carousel, the workload generator is a Python script. BFT-Smart is bundled with a Java client-side library; accordingly, for this system, we write the workload generator in Java.

As mentioned earlier, we place all client nodes in Washington. We use 10 VMs, each hosting

a client. Each client runs the workload generator instantiated with a predefined thread pool size. Depending on the target system and its size, we use between 10 and 180 threads per client to saturate the system and reach peak throughput. Beside the number of threads, the workload generator accepts a few other parameters, notably: IP and port of a system replica (this is the leader in ZooKeeper, etcd, or BFT-Smart; the head of the chain for ChainR; or a random replica for Carousel); the experiment duration (30 seconds by default); or the size of a request (250 bytes in our case).

It is important that clients synchronize their actions. For instance, client threads should coordinate to start simultaneously. Also, we restart and clean-up the service of each system after each experiment, and we also gather statistics and logs. The scripts for achieving this are common among all systems. We use GNU `parallel` [T+11] and `ssh` to coordinate the actions of all the clients. To control the service-side, we use the Upstart infrastructure [Ups].

3.4 Empirical Study

We now present our observations on the performance decay of five SMR protocols. We break this section in two parts: performance (§3.4.1) and stability results (§3.4.2).

As mentioned before, we use 10 clients, each running multiple workload threads. Upon connecting to a system replica, clients allow for a 20 seconds respite to ensure all connections establish correctly. Then all client threads begin the same workload. Each execution runs for 30 seconds (excluding a warm-up and cool-down time of 15 seconds each) and each point in the performance results is the average of 3 executions. For stability experiments we use executions of 60 seconds, with a maximum of up to 120 seconds.

3.4.1 Performance of SMR at 100+ Replicas

We first discuss throughput and then latency. For throughput, we report on the peak value, i.e., throughput when the system begins to saturate, *before* latency surges. We compute this as the sum of the throughput across all 10 clients. We also plot the standard deviation, though often this is negligible and not visible in plots.

When reporting latency, we give the average value at peak throughput, as observed by one of the clients. Since all clients reside in Washington and connect to the same replica, they experience similar latencies. The exception to this is in Carousel, where clients connect to random replicas of the system; to be fair, we present the latency of a client connecting to a replica in Washington.

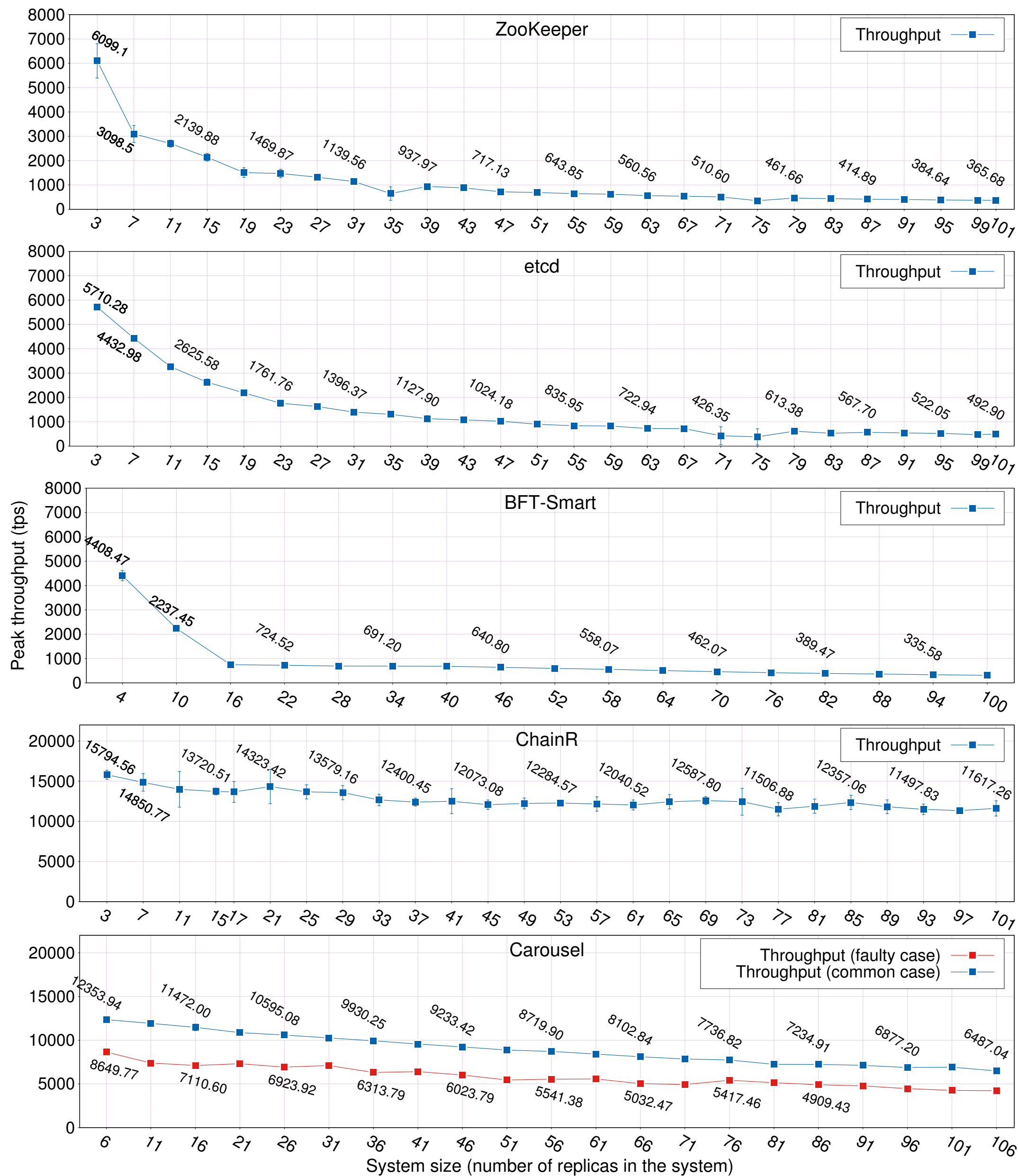


Figure 3.1 – Throughput decay for five SMR systems on a public wide-area cloud platform. For enhanced visibility, we use separate graphs for each system. We also indicate actual throughput values alongside data points. Notice the different axes.

Throughput

Figure 3.1 presents our observations on the throughput decay in five SMR systems, as we increase their size. For readability, we also indicate throughput values on most data points.

We start discussing the three mature systems—ZooKeeper, etcd, and BFT-Smart. For ZooKeeper and etcd, which are CFT, we start from a minimum of 3 replicas and then grow each system in increments of 4 until we reach 101 replicas. For BFT-Smart, we start from 4 replicas—the minimum configuration which offers fault-tolerance in a BFT system—and we use increments of 6 replicas up to $N = 100$. Since we use different increments and start from different system sizes, the x-axes in Figure 3.1 differ slightly across these systems. In terms of fault thresholds, $F = \lfloor \frac{N-1}{2} \rfloor = 50$ for CFT systems, and $F = \lfloor \frac{N-1}{3} \rfloor = 33$ for BFT-Smart.

Roughly speaking, the throughput in ZooKeeper, etcd, and BFT-Smart is inversely proportional to system size N (first three rows in Figure 3.1). Analytically the slopes approximate a $O(1/N)$ function. As we mentioned, these three systems are leader-centric: The leader replica poses a bottleneck by having to disseminate operations to all N nodes. This bottleneck in SMR is well-known [AGK+15, KAD+07, MXC+16], and prior studies reveal this sharp decline in throughput at smaller system sizes [BSA14, CML+06, HKJR10]. If this initial trend would continue, then throughput would quickly decay. What happens, however, is that the trend tapers off. Notably from 20 replicas onward, throughput decays more and more gracefully.

If we look at the throughput function beyond a few tens of replicas, we notice that it follows an exponential decay rate. The sharpest decline is at small system sizes (consistent with earlier observations [CML+06, HKJR10]) but as each system grows, the decay lessens.

We give several intuitions behind the exponential throughput decay rate. First, the higher the throughput of a system is, the more expensive it is to maintain that throughput with a growing system size. For instance, if the throughput is 380 tps (transactions per second) at $N = 95$ replicas and we grow the system by 1 replica, the leader has to send roughly $w \cdot 380$ additional messages (where w is typically 2 or 3, denoting the number of protocol phases). If throughput is $5K$ tps at $N = 3$ and we add 1 additional replica, the leader has to send roughly $w \cdot 5K$ additional messages to sustain the same throughput.

Throughput decay dampens as we grow each system because every additional replica incurs an amount of work depending on the current system size. Adding a replica when $N = 3$ is more costly than adding a replica when $N = 100$, given that there are some fixed processing overheads at the leader which get amortized with system size (e.g., message serialization). We also note that in a larger system there are more tasks (such as broadcast) executing in parallel. Finally and most importantly perhaps, throughput saturates at higher latencies when the systems are larger (see §3.4.1), since the processing pipeline depends on more replicas. In other words, as each system grows, there is a tendency to trade latency for throughput.

We observe that absolute throughput numbers at 100 replicas are in the same ballpark for

these three SMR systems, ranging from 311 to 490 tps. As a side note, this is $45 - 70x$ the current peak theoretical throughput of Bitcoin, suggesting that we can use SMR effectively in mid-sized blockchains, e.g., 100 replicas. If we extrapolate from our observation on the decay rate, it follows that these systems can match Bitcoin peak throughput at about 4500 – 6700 replicas. This is an interesting observation, as the Bitcoin network has about the same size (circa late 2016 [CDE+16]). We interpret this as a simple coincidence, however.

Interestingly, BFT-Smart almost matches the performance of its CFT counterparts. This suggests that BFT SMR protocols decay (in terms of throughput) not much quickly than CFT protocols, regardless of typical quadratic communication complexity of BFT. All three systems show relatively stable performance, and consequently the standard deviations bars are often imperceptible in our plots.

ChainR. We deploy ChainR using the node placement heuristic presented earlier (§3.3.1). The throughput evolution results for this system are in the fourth row of Figure 3.1; please note the y-axis range. As expected, ChainR preserves its throughput very well with increasing system size [AGK+15]. So long as the system stays uniform—i.e., without reducing replica computing performance or bandwidth—the throughput degrades very slowly and at a linear rate. At $N = 101$, the system sustains 73% of the throughput it can deliver when $N = 3$. This is not entirely surprising, as adding replicas to ChainR does not increase the load on any single node (in the first order of approximation), including the leader (i.e., the head replica).

To conclude, chain replication maintains throughput exceptionally well. This system, however, sacrifices availability in the face of faults or asynchrony. Additionally, the chain is as efficient as its weakest link. Indeed, we repeatedly encountered in our experiments cases with zero throughput. Most often, this happened due to a replica crashing, but also in a few cases due to misconfiguration of a successor, therefore leaving the tail node unreachable.

Carousel. This ring-based system can maintain availability despite faulty (or straggler) nodes. Accordingly, we have two measurements: (1) a common case showing the throughput during well-behaved executions, and (2) a sub-optimal (faulty) case when F faults manifest. Since Carousel tolerates up to one-fifth faulty replicas [MA06], we set $F = 21$.

The faulty replicas occupy successive positions on the ring topology, and collaboratively they aim to create a bottleneck in the system. They do so by activating fallback paths on the ring structure. Concretely, they request from the same correct replica—the *target*—to accept traffic from each of them and pass that traffic forward on the ring. The target is the successor of the last faulty node. To make matters worse, the target is also the leader replica (we call this the sequencer, described in §4.1.1). We note that faulty replicas might as well stop propagating updates; but this has a lesser impact on throughput, as the target node would need not process the messages from all faulty nodes. This scenario is among the worst that can happen in terms of throughput degradation to Carousel, barring a full-fledged DDoS attack or a crashed leader (in the latter case progress would halt in any leader-based SMR algorithm).

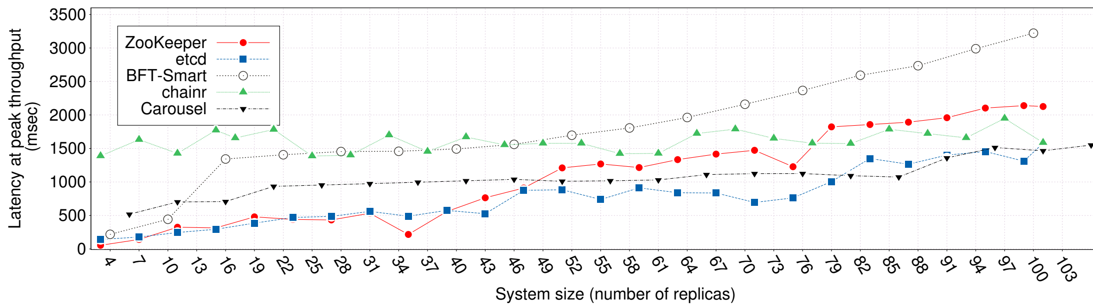


Figure 3.2 – Average latencies (at peak throughput) for experiments with SMR systems.

The results are in the fifth row of Figure 3.1. The throughput decays similarly in the good and faulty cases; in absolute numbers there is a difference of ∼$3k$ tps at every system size. Another way to look at it is that faulty nodes cause on average 30% loss in throughput. Carousel degrades less gracefully than ChainR, as it includes additional BFT mechanism (§4.1.1). Nevertheless, the decay rate in Carousel follows a linear rate, and comes within 60% of ChainR throughput (which we regard as ideal, assuming fault-free executions). In contrast to leader-centric solutions, the chain- and ring-replication systems avoid the bottleneck at the leader and, as expected, exhibit less throughput decay, having more efficient dissemination overlays.

Latency

We present the latency results at peak throughput for all five systems in Figure 3.2. At small scale, CFT systems (ZooKeeper and etcd) exhibit latencies on the order of tens of milliseconds. In contrast, BFT-Smart entails an additional phase (round-trip) in the agreement protocol for every request, which translates into higher latencies. (Batching helps compensate for this additional phase in terms of throughput.)

We remark on the high latency of ChainR. This is to be expected, since this system trades throughput for latency, but it is also amplified by an implementation detail. Specifically, each client runs an HTTP server, waiting for replies from the tail (a distant replica in Sao Paolo). The server is based on the `cherrypy` framework (written in Python, and unoptimized). At low load, the latency in ChainR is similar to Carousel. But in ChainR clients create a larger volume of requests to saturate the system and also run the HTTP server, elevating the load and latency on each client. (In fact, in an earlier version of ChainR, clients were the bottleneck.)

The average latency across all SMR systems does not surpass 3.5 seconds. We only include the latency for the good case of Carousel, but even in the faulty case, latency does not exceed 3.2s. In terms of 99th percentiles, the worst cases are 6.5s for BFT-Smart ($N = 100$), and 6s for the faulty-case of Carousel ($N = 107$).

3.4.2 Stability experiments

Our goal here is to evaluate if mature SMR systems recover efficiently from a serious fault when running at large scale. Concretely, we crash the leader replica (triggering the leader-change protocol) and measure the impact this has on throughput. We cover ZooKeeper, etcd, and BFT-Smart; the other systems (ChainR and Carousel) have no recovery implemented.

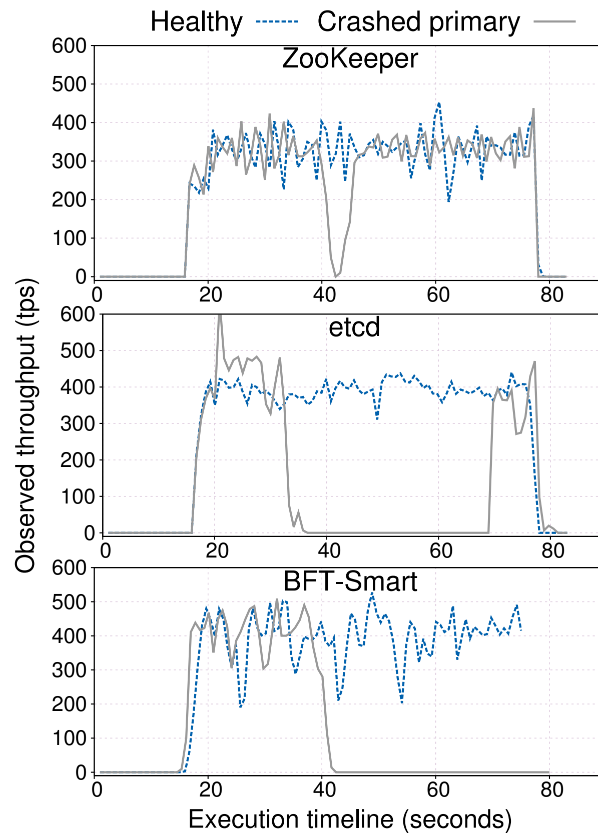


Figure 3.3 – Stability experiments where we crash the primary (i.e., leader), testing the leader-change protocol for ZooKeeper and etcd ($N = 101$), as well as for BFT-Smart ($N = 100$).

When the leader in an SMR system crashes, this kicks off an election algorithm to choose a new leader. We study the stability of this algorithm—whether it works at scale and how much time it requires. These tests are rather about the code maturity in these systems rather than their algorithmic advantages. Our results are in Figure 3.3, describing two runs: (1) a *healthy* case, and (2) a case where we *crash* the leader (i.e., primary). During the first 20s clients simply wait, and then they start their workload; we crash the leader 20s later. The point where we crash the leader is obvious, around 40s, as the throughput drops instantly to zero.

ZooKeeper has consistently the fastest recovery. The throughput reaches its peak within just a few seconds after the leader crashes, consistent with earlier findings at smaller scale [SRMJ12]. This system has an optimization so that the new leader is a replica with the most up-to-date state, which partly accounts for fast recovery [Ong14]. In etcd recovery is slower: It can take

up to 40 seconds for throughput to return to its peak. The election mechanism in etcd is similar to that of ZooKeeper [Ong14], and the heartbeat parameters of these two systems are similar as well (1 and 2 seconds, respectively). The difference in stability between ZooKeeper and etcd can also stem from an engineering aspect, as the former system has a more stable codebase (started in 2007) compared with the latter system (started in 2013).

For BFT-Smart, we allowed the system up to 120 seconds to recover but throughput remained 0. We also tried with smaller sizes, and found that $N = 88$ is the largest size where BFT-Smart manages to recover, after roughly $10s$. Finally, we remark that we were able to reproduce all these behaviors across multiple (at least 3) runs, so these are not outlying cases.

4 Carousel: Mitigating Throughput Decay in State Machine Replication

and all alone
a drop in a waterfall dropping
falls into an eternal flow

— Vlado Škafar [Šk15]

In this chapter we present the Carousel SMR system. We first describe the common-case protocol (§4.1), and then discuss the reconfiguration algorithm (§4.2). We defer the correctness discussion of Carousel to the appendix of this dissertation (Appendix A).

An interesting design aspect of Carousel is how the ring overlay masks faults. We achieve this by keeping redundant paths on the ring overlay, thus ensuring availability despite faults or asynchrony. We believe of equal interest is also the agreement protocol of Carousel, which is essentially the FaB consensus algorithm of Martin and Alvisi [MA06] adapted to a ring overlay.

We choose to pattern the agreement protocol of Carousel after FaB due to the interesting tradeoff this offers. FaB reduces the latency of BFT agreement from three steps to two [MA06]. This is appealing in our ring-based topology, because each step entails a complete traversal of the ring. Fewer traversals means higher throughput, lower latency, and a simpler protocol. This benefit comes to the detriment of resilience, however: The system needs larger quorums to tolerate faults. Carousel assumes $N = 5F+1$ replicas, whereas optimal BFT systems tolerate one-third faults, i.e., $N = 3F + 1$ [CL02]. As in prior solutions, our agreement protocol relies on the existence of a *sequencer* (i.e., a leader) assigning sequence numbers to operations.

4.1 Carousel Common-Case Protocol

Figure 4.1 shows an overview of Carousel in a system of $N = 6$ replicas. The replicas, labeled from 0 to 5, are organized in a ring overlay. Note that each replica in this overlay has a certain *successor* and *predecessor* replica. One of the replicas (node 0) is the sequencer. By default, broadcast messages disseminate throughout the whole system from one replica to its imme-

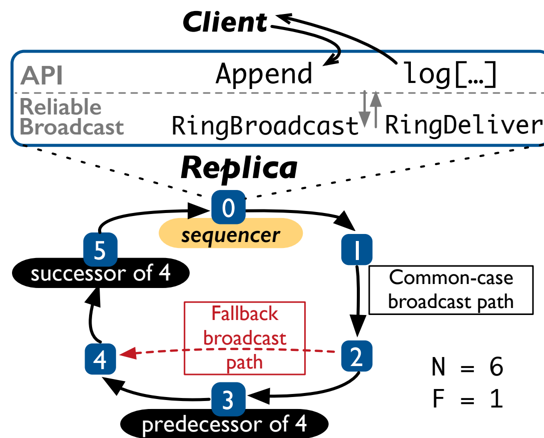


Figure 4.1 – Overview of Carousel in a system of 6 replicas. Clients interact with the system via a thin API. Underlying this API, there is a reliable broadcast scheme executing along a ring overlay network. In this overlay, each replica has a *successor* and *predecessor*. There is a specific replica which carries the role of a *sequencer* (replica 0 here).

diate successor. Additionally, fallback (redundant) paths exist, to ensure availability.

The interface of Carousel is log-based: Each replica exposes a simple API allowing clients to read from a totally-ordered *log*, and to *Append* new entries to this log. Under this API, all replicas implement a reliable broadcast primitive providing high throughput and availability. We discuss the Append operation first (§4.1.1), and then the reliable broadcast layer (§4.1.2).

4.1.1 Append Operation

To add an entry e to the totally-ordered log, clients invoke Append(e) at any replica i. The operation proceeds in two logical phases, as follows.

1. **Data**—Replica i broadcasts entry e to all correct replicas using the *RingBroadcast* primitive of the underlying broadcast layer.

2. **Agreement**—A BFT agreement protocol executes. The sequencer proposes a sequence number (i.e., a log position) for entry e, and correct replicas confirm this proposal. After executing the agreement phase for this entry, replica i notifies the client that the operation succeeded.

Listing 4.1 shows the implementation of the Append operation. First, replica i broadcasts a $\langle$DATA, $e\rangle$ message, as shown on line 2. This corresponds to the first logical phase of the Append operation. We say that this broadcast message is of *type data* and has *payload e*. As this message disseminates throughout the system, each correct replica triggers the *RingDeliver* callback to deliver the data message (line 4 of Listing 4.1).

```
1 func Append(e):
2   RingBroadcast(⟨DATA, e⟩) // Disseminates a data message with payload 'e'

4 callback RingDeliver(id, ⟨type, payload⟩):
5   if (type == DATA):
6     pending[id].entry = payload
7     proposeAgreementMsg(id, payload) // Executes ONLY at sequencer
8   else if (type == AGREEMENT):
9     handleAgreementMsg(payload)   // The payload is a triplet

11 // Executes ONLY at the sequencer
12 func proposeAgreementMsg(id, e):
13   hash = Hash(nextSeqNr . id . e)
14   // Disseminate the agreement message
15   RingBroadcast(⟨AGREEMENT, nextSeqNr++, id, hash⟩)

17 func handleAgreementMsg(sn, id, hash):
18   validate(sn, id, hash) || return
19   pending[id].sn   = sn
20   pending[id].hash = hash
21   pending[id].confirmations++
22   // Disseminate our own confirmation for this sequence number
23   RingBroadcast(⟨AGREEMENT, sn, id, hash⟩)
24   if (pending[id].confirmations == 4F+1):
25     addToStableLog(id)
```

Listing 4.1 – A high-level algorithm describing the Append operation in Carousel.

The delivery callback always provides two arguments: (1) an identifier *id*, and (2) the actual message. The underlying broadcast layer assigns the *id*, which uniquely identifies the associated message. We discuss identifiers in further detail later, but suffice to say that an identifier is a pair denoting the replica which sent the corresponding message plus a logical timestamp for that replica (§4.1.2). The actual message, in this case, is a data message with payload entry *e*. Upon delivery of any data message, each correct replica stores this entry in a *pending* set. Note that this set is indexed by the assigned *id* (line 6).

The agreement phase starts when the sequencer replica delivers the data message with *e*. After saving *e* in the pending set, the sequencer also proposes a sequence number for *e* by broadcasting a $\langle$AGREEMENT, $sn, id, hash\rangle$ message (line 7 and lines 12–15). It would be wasteful (in terms of bandwidth) to include the whole entry in this agreement message; instead, the sequencer simply pairs the entry *id* with a monotonically increasing sequence number *sn* (called *nextSeqNr* on line 15). The *hash* in the agreement message is computed on the concatenation of the assigned sequence number, the entry *id*, and the entry *e* itself.

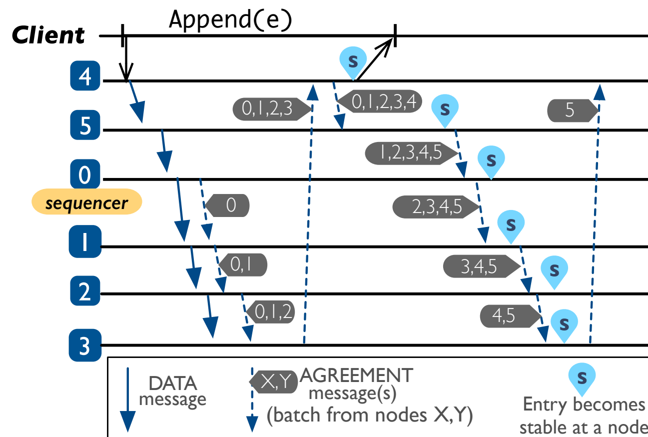


Figure 4.2 – The unfolding of an Append(e) operation in Carousel. The client contacts replica 4, which broadcasts a data message to all replicas. Once this message reaches the sequencer (node 0), this replica broadcasts an agreement message. Thereafter, all replicas broadcast their own agreement message. The entry e becomes stable once a replica observes at least $4F+1=5$ matching agreement messages for that entry.

Each replica delivers the agreement message of the sequencer trough the same RingDeliver callback of the broadcast layer. After any replica j delivers the agreement message, then j broadcasts its own agreement message with the same triplet (*sn, id, hash*), as described on line 23. Prior to doing so, replica j validates the hash and saves the assigned sequence number and hash in the pending log (lines 17–20).

The validation step (line 18) refers to several important checks: that the *hash* correctly matches the sequence number *sn*, identifier *id*, and entry content; that this *id* has no other proposal for a prior sequence number; that each number *sn* in an agreement message has a corresponding proposal for that *sn* from the sequencer replica; and that different agreement messages confirming the same *sn* originate from distinct replicas.

We say that a replica commits on log entry *e* after it gathers sufficient ($4F+1$) confirmations for that entry. The entry then becomes stable at that replica (line 25). By the reliability of the RingBroadcast dissemination primitive, if the sequencer is correct and proposes a valid sn, then the entry eventually becomes stable at all correct replicas. As proved in prior work [MA06], this protocol has optimal resilience for two-step BFT agreement, i.e., $N=5F+1$ is the smallest system size to tolerate F faults with two-step agreement.

Informally, as the initial agreement message (from the sequencer) propagates on the ring, it produces a snowball effect: Each replica delivering this message broadcasts its own agreement message, confirming the proposed sequence number. Figure 4.2 depicts this intuition.

4.1.2 Reliable Broadcast in a Ring Overlay

In a conventional ring-based broadcast, each replica i expects its predecessor i-1 to forward each message which i-1 delivers [GLPQ10, JMPSP17]. Messages travel from every replica to that replica's successor. Intuitively, this scheme is throughput-optimal because it balances the burden of data dissemination across all replicas [GLPQ10]; the downside, however, is that asynchrony (or fault) at *any* replica can affect availability by impeding progress.

To maintain high-throughput dissemination across the ring overlay despite asynchrony or faults, in Carousel we strengthen the ring so that each replica connects with $F+1$ total predecessors. A replica has one *default* connection—with the immediate predecessor—and up to F *fallback* connections—with increasingly distant predecessors. The topology we obtain is essentially an F-connected graph. This graph ensures connectedness (availability) despite up to F faults.

In Figure 4.1 for instance, replica 4 should obtain from replica 3 all messages circulating in the system. If replica 3 disrupts dissemination and drops messages, however, then replica 4 can activate the fallback connection to replica 2. By default, communication on this fallback path is restricted to brief messages called *state vectors*, which replica 2 periodically sends directly to replica 4.

More generally, replica i expects a state vector from all F of its fallback predecessors, i.e., from replicas i-2, i-3, ... A state vector is a concise representation of all messages delivered by the corresponding fallback replica. If replica i notices that its immediate predecessor i-1 is omitting messages (and that the state on fallback replica i-2 is steadily growing), then i sends a $\langle$ACTIVATE, $sv_i\rangle$ message directly to replica i-2, where sv_i is the state vector of replica i.

Replica i-2 interprets the activate message by sending to replica i all messages which i-2 has delivered and are not part of state vector sv_i. Thereafter, replica i-2 continues sending to i any new messages it delivers. In the meantime, replica i-2 also continues forwarding messages as usual to its immediate successor replica i-1. In case replica i-1 restarts acting correctly and forwards messages to i, then replica i can send a $\langle$DEACTIVATE$\rangle$ message to i-2.

Alternatively, replica i can request individual pieces of the state from i-2, e.g., in case replica i-1 is selectively withholding messages from i. Since every replica has $F+1$ total connections, Carousel can tolerate up to F faults, regardless whether these faults are successive on the ring or dispersed across the system. This mechanism based on fallback connections is strictly to improve availability (i.e., delivery of broadcast messages) relying on timeouts, but does not affect the safety of the protocol (Appendix A.1).

The state vector sv_i at some replica i is a vector of timestamps with one element per replica. The element on position j denotes the latest message broadcast by replica j which replica i delivered. Concretely, each such element is a timestamp, i.e., a logical counter attached to any message which a replica sends upon invoking RingBroadcast. For instance, whenever

replica j calls RingBroadcast(m), the broadcast layer tags message m with a unique *id*, in the form of a pair $\{j, ts\}$, where j denotes the sender replica and ts is a monotonically increasing timestamp specific to replica j. [1] As we explained earlier, the *id* also plays an important role in the Append operation (§4.1.1).

When replica i delivers message m with id $\{j, ts\}$ from replica j via the RingDeliver callback, replica i updates its state vector sv_i to reflect timestamp ts for position j. Status vectors are inspired from vector clocks [Fid88, Mat88]. The notable difference to vector clocks is that a replica does not increment its own timestamp when delivering or forwarding a message: This timestamp increments only when a replica sends a new message—typically a data or a agreement message—by invoking RingBroadcast. With state vectors, our goal is not to track causality or impose an order [GLPQ10], but to ensure no messages are lost (i.e., reliability).

One corner-case that can appear in our protocol is when a malicious sender replica attempts to stir confusion using incorrect timestamps. In particular, such a replica i can attach the same timestamp to two different messages: $m1$ with $\{i, ts1\}$ and $m2$ with id $\{i, ts1\}$. Another bad pattern is skipping a step in the timestamp by broadcasting first $\{i, ts1\}$ and then $\{i, ts3\}$. In practice, communication between any two replicas relies on FIFO links (e.g., TCP), so correct replicas can simply disallow—as a rule—gaps or duplicates in messages they deliver.

FIFO links, however, do not entirely fix the earlier problem. It is possible that some replica delivers $m1$ with timestamp $ts1$, while a different replica delivers $m2$ for this same timestamp. But note that this will not cause safety issues. The replicas can only agree on either of $m1$ or $m2$: When the sequencer proposes a sequence number for id $\{i, ts1\}$, it also includes a hash of the corresponding message, either $m1$ or $m2$. Even if the sequencer is incorrect and proposes sequence numbers for *both* $m1$ and $m2$ (a poisonous write [MA06]) only one of these two messages can gather a quorum and become stable. To conclude, timestamps restrict acceptable behavior and—when coupled with the sender replica's identity—provide a unique identifier for all messages circulating in the system, which helps ensure reliability.

4.2 Carousel Reconfiguration

The reconfiguration sub-protocol in Carousel ensures liveness by changing the sequencer when this replica misbehaves. This protocol is, informally, an adaptation of the FaB [MA06] recovery mechanism (designed for an individual instance of consensus) to SMR.

4.2.1 Preliminaries

Reconfiguration concerns the agreement algorithm in Carousel. Neither the data phase (of the Append operation) nor the broadcast layer need to change. We adjust the common-case

[1] Just like sequence numbers, timestamps are *dense* [CL02]: This prevents replicas from exhausting the space of these numbers and makes communication steps more predictable, which simplifies dealing with faulty behavior [AAC+05].

protocol of §4.1.1 to accommodate reconfiguration as follows:

1. Replicas no longer agree on a sequence number sn for every Append operation. Instead, each instance of the agreement protocol for a given sequence number is tagged with a *configuration number*, so that replicas now agree on a tuple $\{cn, sn\}$. In other words, agreement messages now have the form $\langle$AGREEMENT$, \{cn, sn\}, id, hash\rangle$. The concept of *proposal numbers* in FaB [MA06] or that of *view numbers* in PBFT [CL02] is analogous to configuration numbers in Carousel; briefly, these serve the purpose of tracking the number of times the sequencer changes.
2. We modify the $hash$ in agreement messages to also include the configuration number.
3. Our common-case protocol assumes that, for every sequence number sn, a replica only ever accepts (i.e., gives its vote for) a single agreement message, namely, the first valid agreement message they deliver for that sn from the sequencer. To account for configuration numbers, a replica is now allowed to change its mind, and accept another agreement message if the sequencer changed (i.e., in a different configuration number).

Configuration numbers in Carousel start from 0. In FaB, during a reconfiguration, the new sequencer is elected using a separate leader election protocol. In Carousel, we do not rely on a leader election protocol; instead, every time the configuration number increases, the sequencer role changes deterministically, so that the new sequencer is the successor of the previous sequencer, in a round robin manner.

We note that the ring-based broadcast algorithm (§4.1.2) that replicas employ during common-case requires no modification due to reconfiguration. While executing reconfiguration (described next), however, replicas do not use the ring-based broadcast primitive. If reconfiguration is executing, this means that the current sequencer is faulty and hence the system is experiencing no progress. For this reason, we can temporarily renounce on the high-throughput broadcast enabled by the ring topology, and adopt instead a conventional all-to-all broadcast scheme towards optimizing for latency.

4.2.2 Carousel Reconfiguration Protocol

A correct replica i enters the reconfiguration sub-protocol if any of these two conditions hold: (1) a timer expires at replica i because the sequencer replica failed to create new agreement messages or a previous reconfiguration failed to complete in a timely manner; or (2) replica i observes $F+1$ other replicas proposing reconfiguration.

To propose a reconfiguration and change the sequencer, replica i broadcasts a $\langle$RECONF, i, cn, $\mathscr{P}\rangle$ message. Once it does so, replica i also starts ignoring any messages concerning configuration number cn or lower, and until reconfiguration completes it ignores any messages except those of type data, reconfiguration, or new-configuration (as we define them below). Replica i also starts a timer to prevent a stalling reconfiguration.

The set $\mathscr{P}$ in a reconfiguration message contains agreement messages which replica i signed for all the sequence numbers which are still pending at this replica. In other words, these are proposals for entries which are not part of the stable log at replica i because this replica gathered insufficient votes (i.e., less than $4F+1$) to mark the corresponding entry as stable.

The successor of the faulty (old) sequencer, namely the replica at position cn' on the ring, where $cn' = (cn+1)\%N$, is set to become sequencer when reconfiguration completes. This new sequencer waits until it gathers reconfiguration messages from $4F+1$ replicas, including itself, and then broadcasts a $\langle$NEWCONF, $cn+1$, $\mathscr{R}\rangle$ message. Here, $\mathscr{R}$ represents the set of $4F+1$ reconfiguration messages which the new sequencer gathered.

The reconfiguration sub-protocol completes at a replica i when that replica delivers the new-configuration message. When this happens, replica i starts accepting agreement messages created by the new sequencer (i.e., the replica at position cn' on the ring) and expects these messages to be tagged with configuration number $cn+1$. After reconfiguration completes, the new sequencer starts redoing the common-case agreement protocol for every individual sequence number found in an agreement message in the set $\mathscr{R}$.

For every sequence number sn appearing in $\mathscr{R}$, the new sequencer creates a new agreement message with the configuration number set to $cn+1$. When creating these new agreement messages, for every sequence number sn there are two cases to consider:

1. If there exists an id that appears in an agreement message of $\mathscr{R}$, such that there is no $id' \neq id$ with $2F+1$ agreement messages (with the same sn) in $\mathscr{R}$, then the new sequencer chooses this id to be associated to sn in the new agreement message. The new sequencer also recomputes the hash as done in the common-case protocol, including the new configuration number $cn+1$. In FaB terminology [MA06], we say that the set $\mathscr{R}$ of reconfiguration messages *vouches* for id to be associated to sn.

2. Alternatively, it can happen that more than one pair (sn, id) is vouched for by $\mathscr{R}$ (or no such pair at all). This can occur, for instance, if the previous sequencer was malicious and proposed for the same sequence number sn multiple different ids. In this case, the sequencer can choose to associate sn with any id that was previously proposed for sn, and recomputes the hash as done in the common-case.

For every such new agreement message that the new sequencer creates, the common-case protocol executes as described in §4.1.1, accounting for the configuration number $cn+1$.

5 Discussion & Concluding Remarks

This chapter comprises a general discussion related to performance decay in SMR and to Carousel (§5.1), as well as our concluding remarks for this first part of the dissertation (§5.2).

5.1 Discussion

Most popular SMR protocols, including those we study in this chapter, have been designed, implemented, and evaluated assuming uniform replica and network characteristics [Bur06, Lam98, Ong14]. In a WAN (or multi-datacenter), uniformity is unlikely. Yet systems in the spirit of chain- or ring-replication excel in uniform settings [GLPQ10]; such systems, however, forfeit availability otherwise. For instance, in non-uniform networks if there are stragglers or asynchrony this impedes progress. Note that such conditions do not affect availability in leader-centric (i.e., star overlay) protocols. Clearly, SMR protocols face a tangible design tradeoff between their common-case performance and their availability.

Ideally, SMR systems should achieve a middle-ground between performance (i.e., efficient dissemination and agreement) and high availability, being capable of graceful degradation in the face of faults or stragglers. This was one of our goals in Carousel.

We remark on two concrete research directions to help further the goal of reconciling performance and availability in SMR. First, it is appealing to combine broadcast with ring-based protocols in a single system, in the vein of Abstract [AGK+15]. Such a system could provide higher BFT resilience, namely tolerate $F = \lfloor (N-1)/3 \rfloor$ faults, unlike Carousel where resilience is $F = \lfloor (N-1)/5 \rfloor$. Combining protocols typically yields intricate systems, however, and it is important to address the resulting complexity [AGK+15]. Second, for predictable behavior, faulty replicas should be detected and evicted from the system in a timely manner. This is challenging in a Byzantine environment. Past approaches rely on proofs-of-misbehavior (which are useful for limited kinds of faults [KAD+07]) or on incentives (which tend to be complex [AAC+05]), so new solutions or practical assumptions are needed.

For the sake of clarity, we omitted cryptographic primitives from the earlier description of

Carousel. Such a mechanism is orthogonal and is a well-known technique in BFT replication [CL02, BSA14]. We apply this technique in Carousel as follows: Whenever a replica sends a message (e.g., a agreement message or a batch of them), it piggybacks a digest of that message and a signature of the digest. With regards to garbage collection, we implement a sub-protocol similar to the classic checkpoint mechanism of PBFT [CL02, §4.4]; this sub-protocol, however, is out of the scope of this thesis.

Chain- or ring-replication is a well-studied scheme in SMR for avoiding the bottleneck at the leader. Our baseline, ChainR, is a straightforward implementation of chain replication in the fail-stop model, shedding light on what is an ideal upper bound of SMR throughput decay in fault-free executions. In contrast to Carousel, previous approaches assume a model that does not tolerate Byzantine faults [ANRST05, AFK+09, JMPSP17], or they degrade to a broadcast algorithm to cope with such faults [AGK+15, Kne12]. Carousel retains the efficient broadcast overlay based on a ring topology despite Byzantine replicas.

The technique of overlapping groups of chain replication on a ring topology, as employed in FAWN [AFK+09], is similar to ring-based replication. The insight is similar: Instead of absorbing all client operations through a single node (a bottleneck), accept operations at multiple nodes. The most important distinction between FAWN and Carousel lies in the use of sharding. FAWN shards the application state, each shard mapping to a chain replication group. As we mentioned earlier, sharding is a common workaround to scale SMR [BP16, GBKA11, CDE+13]. In Carousel and other systems we study, the goal is *full* replication. Even when sharding is employed, our findings are valuable because they apply to the intra-shard protocol (which is typically an SMR instance).

S-Paxos [BMSS12] decouples request *dissemination* from request *ordering*. This relaxes the load at leader and can increase throughput. We apply this same principle in Carousel, by separating dissemination (ring-based broadcast for high throughput) from agreement (FaB). In contrast to Carousel, S-Paxos does not tolerate Byzantine faults.

Building upon a conjecture of Lamport [Lam03], FaB laid the fundamental groundwork for two-step BFT consensus [MA06]. The agreement algorithm in Carousel is a simplified FaB protocol (e.g., we use only one type of protocol message, AGREEMENT, for reaching agreement). But the most important distinction to FaB is that Carousel employs a ring topology in the common-case, eschewing the throughput bottleneck at the leader. We believe the FaB agreement algorithm combined with efficient broadcast schemes from the chain- or ring-based families [GLPQ10, JMPSP17, VRS04] deserve more attention.

It was recently uncovered that a version of the FaB protocol is flawed. Namely, that this protocol version suffers from liveness issues, which can happen when a malicious leader engages in a poisonous write [AGM+17]. This problem, however, only applies specifically to the parametrized version of FaB [MA06], called PFaB in [AGM+17]. PFaB is not the same protocol as the one we use in Carousel, hence Carousel is not subject to these liveness problems.

An important goal in Carousel was simplicity, an often understated property of SMR systems, and an important obstacle to their practical adoption [Ong14]. A tree overlay could potentially provide a better latency/throughput tradeoff than a ring, but would do so at significant added complexity and potential safety issues [AMN+18, KJG+16].

Our stability experiments show that BFT SMR protocols, even as mature as BFT-Smart, are not as battle-tested as CFT protocols. Indeed, open-source implementations of BFT SMR are scarce, and the issue of *non-fault-tolerant* BFT protocols is known [CWA+09]. In fact, it is a pleasant surprise for us that BFT-Smart is able to go through reconfiguration at 88 replicas (despite folding at bigger system sizes), and to resume request execution after the leader fails.

An important application of BFT SMR is for permissioned digital trust (e.g., distributed ledger) applications [ABB+18, SBV18]. In such an application, SMR can serve as an essential component ensuring total-order across the operations in the system. There is a growing body of work dealing with the problem of scaling consensus for these applications, which we cover briefly below.

A practical way to scale agreement to a large set of nodes is to elect a small committee (or even a single node), and run the expensive agreement protocol in this committee. The latest in this line of research is Algorand [GHM+17], which relies on a novel Byzantine consensus protocol called $BA\star$. Other notable efforts include HoneyBadger [MXC+16], ByzCoin [KJG+16], or Bitcoin-NG [EGSVR16]. Most of the research in this area seeks to ensure probabilistic guarantees, whereas throughout this dissertation we consider SMR systems providing deterministic guarantees.

HoneyBadger and ByzCoin show impressive results in their experiments (throughput of $7k$ tps and $1k$ tps, respectively) while running on 100+ nodes, albeit with large batches (8MB). Similarly, Algorand exhibits more than a hundred-fold throughput improvement over Bitcoin (i.e., they achieve ~700 tps) when scaled to 50,000 participants. A further caveat in these systems is that they rely on a cryptocurrency. The SMR systems we study, in contrast, are more generic—being cryptocurrency-free—but also more strict in their trust model—because they assume a permissioned system where the set of participants is regulated.

A specific step that is common to all SMR protocols is the processing of protocol messages and executing requests at each replica. Several approaches can optimize this step. These are all orthogonal to our study, and they generally apply to any SMR system. Examples include optimistic execution to leverage multi-cores [KWQ+12] or hardware-assisted solutions [PH15], e.g., to speed costly crypto computations, offloading protocol processing [ISAV16], or using a trusted module to increase resilience and simplify protocol design [CMSK07].

5.2 Conclusions

It is commonly believed that SMR throughput degrades sharply as system size grows. The data supporting this belief is scarce (for up to a couple of tens of nodes), however, and simple extrapolation cannot tell the whole story. In this chapter we filled in this missing information by providing empirical observations on how SMR performance decays with system size.

Our experiments covered different design dimensions in SMR systems: three failure models (asynchronous CFT, fail-stop CFT, and BFT); different failure thresholds; three types of network overlays (star, chain, and ring). The systems are also varying in their implementation maturity, from production-ready codebases to research prototypes.

We provide two broad takeaways from our performance decay evaluation. First, we noticed that in leader-centric protocols (i.e., star overlays), the performance decay follows an exponential decay rate, being very sharp at the beginning and slowing down as systems grow beyond a few tens of nodes. Second, we saw that chain or ring overlays can mitigate the exponential decay rate; these systems, however, require additional techniques to remain available and deal with asynchrony or faults, indicating the next relevant challenges for future work.

For the three mature SMR systems we considered (ZooKeeper, etcd, and BFT-Smart), we also provided empirical results on their stability with regards to the leader-change protocol. We found that ZooKeeper can recover very fast from a crashed leader, being the most stable system we studied.

We believe our observations in this chapter help inform future designs of SMR algorithms. Our study has several limitations, however, which can be addressed in future work as well. To fundamentally understand performance decay in SMR, a more comprehensive array of experiments is necessary, for instance by including a persistence layer on the critical path of ordering, going beyond 100 replicas, doing evaluation on different platforms, etc. There are also economic factors at play, since platforms for running these experiments do not come for free. For instance, our experiments cost on the order of thousands of dollars. It would be helpful to develop tools, methods, and platforms for lowering these costs.

Part III

Supporting Efficient Access to Replicated Data

6 Incremental Consistency Guarantees

We can often enhance our ability to deal with a problem by adopting a language that enables us to describe the problem in a different way.

— Abelson and Sussman [AS84]

In this chapter we introduce Correctables, an abstraction that provides support for programming with replicated objects. The goal of this abstraction is to hide most of the complexity underlying replication, allowing developers to focus on the task of balancing consistency and performance. To aid developers with this task, Correctables provide *incremental consistency guarantees*, which capture successive refinements on the result of an ongoing operation on a replicated object. In short, applications receive both a preliminary—fast, possibly inconsistent—result, as well as a final—consistent—result that arrives later.

We show how to leverage incremental consistency guarantees by speculating on preliminary values, trading throughput and bandwidth for improved latency. We experiment with two popular storage systems (Cassandra and ZooKeeper) on the Amazon EC2 platform, showing how to hide the latency of strongly consistent operations by up to 40%.

6.1 Introduction

Replication is a crucial technique for achieving performance—i.e., high availability and low latency—in large-scale applications, as we discussed earlier in this dissertation (§1.1). Traditionally, strong consistency protocols hide replication and ensure correctness by exposing a single-copy abstraction over replicated objects [CDE+13, Lam98]. There is a tradeoff, however, between consistency and performance [Aba12, Bre12, GL02]. In the quest toward more efficient algorithms and better performance, many systems choose to employ weak consistency models [DHJ+07]. Unfortunately, this choice introduces the possibility of incorrect (anomalous) behavior in applications.

A common argument in favor of weak consistency is that such anomalous behavior is rare in

practice. Indeed, studies reveal that on expectation, weakly consistent values are often correct even with respect to strong consistency [BVF+12, LVA+15]. Applications which primarily demand performance thus forsake stronger models and resort to weak consistency [ABK+15, DHJ+07].

There are cases, however, where applications often diverge from correct behavior due to weak consistency. As an extreme example, an execution of YCSB workload A [CST+10] in Cassandra [LM10] on a small $1K$ objects dataset can reveal stale values for 25% of weakly consistent read operations (Figure 6.7 in §6.6). This happens when using the *Latest* distribution, where read activity is skewed towards popular items [CST+10]. In other cases, even very rare anomalies are unacceptable (e.g., when handling sensitive data such as user passwords), making strongly consistent access a necessity. For this class of applications, correctness supersedes performance, and strong consistency thus takes precedence [CDE+13].

There is also a large class of applications which do not have a single, clear-cut goal (either performance or correctness). Instead, such applications aim to satisfy *both* of these conflicting demands. These applications fall in a *gray zone*, somewhere in-between the two previous classes, as we highlight in Figure 6.1. Typically, these applications aim to strike an optimal balance of consistency and performance by employing different consistency models, often at the granularity of individual operations [BFF+15, CRS+08, KHAK09, LPC+12, TPK+13]. Choosing the appropriate consistency model, even at this granularity, is hard, and the result is often suboptimal, as developers still end up with fixing a certain side of the consistency/performance tradeoff (and sacrificing the other side).

Moreover, programming in the gray area is difficult, as developers have to juggle different consistency models in their applications [CRS+08, KHAK09]. If programming with a single consistency model (such as weak consistency [CDE+13]) is non-trivial, then mixing multiple models is even harder [LLC+14]. In their struggle to optimize performance with consistency, developers must go up against the full complexity of the underlying storage stack. This includes choosing locations (cache or backup or primary replica), dealing with coherence and cache-bypassing, or selecting quorums. These execution details reflect as a burden on devel-

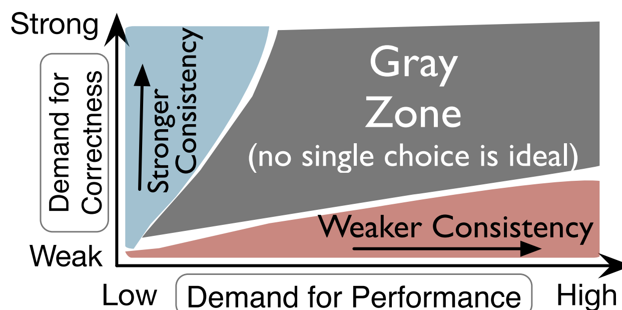


Figure 6.1 – Many applications fall into a *gray zone*, torn between the need for both performance and correctness.

opers, complicate application code, and lead to bugs [Eri13, LVA+15].

Our goal is to help with the programming of applications located in the gray area. We accept as a fact that no single consistency model is ideal, providing both high performance and strong consistency (correctness) at the same time [Aba12, GL02]. Our insight is to approach this ideal in complementary steps, by *combining consistency models in a single operation*. Briefly, developers can invoke an operation on a replicated object and obtain multiple, incremental *views* on the result, at successive points in time. Each view reflects the operation result under a particular consistency model. Initial (preliminary) views deliver with low latency—but weak consistency—while stronger guarantees arrive later. We call this approach *incremental consistency guarantees* (ICG).

We introduce Correctables, an abstraction which grants developers a clean, consistency-based interface for accessing replicated objects, clearly separating semantics from execution details. This abstraction reduces programmer effort by hiding storage-specific protocols, e.g., selecting quorums, locations, or managing coherence. Correctables are based on *Promises* [LS88], which are placeholders for a single value that becomes available in the future. Correctables generalize Promises by representing not a single, but multiple future values, corresponding to incremental views on a replicated object.

To the best of our knowledge, our abstraction is the first which enables applications to build on ICG. As few as *two* views suffice for ICG to be useful. The advantage of ICG is that applications can speculate on the preliminary view, hiding the latency of strong consistency, and thereby improving performance [WCN+09]. Speculating on preliminary responses is expedient considering that, in many systems, weak consistency provides correct results on expectation [BVF+12, LVA+15].

Speculation with ICG is applicable to a wide range of scenarios. Consider, for instance, that a single application-level operation can aggregate multiple—up to hundreds of—storage-level objects [ABK+15, DB13, LMNR15, Sch15]. Since these objects are often inter-dependent, they can not always be fetched in parallel. With ICG, the application can use the fast preliminary view to speculatively prefetch any dependent objects. By the time the final (strongly consistent) view arrives, the prefetching would also finish. If the preliminary result was correct (matching the final one), then the speculation is deemed successful, reducing the overall latency of this operation.

Alternatively, ICG can open the door to exploiting application-specific semantics for optimizing performance. Imagine an application requiring a monotonically increasing counter to reach some predefined threshold (e.g., number of purchased items in a shop required for a fidelity discount). If a weakly consistent view of the counter already exceeds this threshold, the application can proceed without paying the latency price of a strongly consistent view.

The high-level abstraction centered on consistency models, coupled with the performance benefits of enabling speculation via ICG, are the central contributions of Correctables. We

evaluate these performance benefits by modifying two well-known storage systems (Cassandra [LM10] and ZooKeeper [HKJR10]). We plug Correctables on top of these, build three applications (a Twissandra-based microblogging service [Twi19], an ad serving system, and a ticket selling system), and experiment on Amazon EC2.

Our evaluation first demonstrates that there is a sizable time window between preliminary and final views, which applications can use for speculation. Second, using YCSB workloads A, B, and C, we show that we can reduce the latency of strongly consistent operations by up to 40% (from $100ms$ to $60ms$) at little cost (10% bandwidth increase, 6% throughput drop) in the ad system. The other two applications exhibit similar improvements. Even if the preliminary result is often inconsistent (25% of accesses), incremental consistency incurs a bandwidth overhead of only 27%.

In the rest of this chapter, we give an overview of our solution in the context of related work (§6.2) and present the Correctables interface (§6.3). We show how applications use Correctables (§6.4), and describe the bindings to various storage stacks (§6.5). We then give a comprehensive evaluation (§6.6) and conclude (§6.7).

6.2 Overview & Related Work

At a high-level, we address the issue of programming with replicated objects through a novel abstraction called Correctables. We now present the main concepts behind this abstraction while contrasting this approach with prior work in this area.

6.2.1 Consistency Choices

There is an abundance of work on consistency models. These range from strong consistency protocols [JRS11, Lam98, VRS04], some optimized for WAN or a specific environment [CDE+13, DNN+15, KPF+13, LPK+15, XSK+14, ZSS+15], through intermediary models such as causal consistency [DIRZ14, LFKA13], to weak consistency [DHJ+07, TTP+95]. As a recent development, storage systems offer multiple—i.e., *differentiated*—consistency guarantees [CRS+08, KHAK09, PAA+15]. This allows applications in the above-mentioned gray zone to balance consistency and performance on a per-operation basis: the choice of guarantees depends on how sensitive the corresponding operation is.

Differentiated guarantees can take the form of SLAs [TPK+13], policies attached to data [KHAK09], dynamic quorum selection for quorum-based storage systems such as Dynamo [DHJ+07] or others [LM10, Ria19], or even ad-hoc operation invariants [BFF+15]. In practice, two consistency levels often suffice: weak and strong [App18, Sim19]. Sensitive operations (e.g., account creation or password checking) use the strong level, while less critical operations (e.g., remove from basket) use weak guarantees [KHAK09, TPK+13, YV00] to achieve good performance.

For instance, in Gemini [LPC+12], operations are either Blue (fast, weakly consistent) or Red

(slower, strongly consistent). For sensitive data such as passwords, Facebook uses a separate linearizable sub-system [LVA+15]. Likewise, Twitter employs strong consistency for "certain sets of operations" [Sch14], and Google's Megastore exposes strong guarantees alongside read operations with "inconsistent" semantics [BBC+11]. Another frequent form of differentiated guarantees appears when applications bypass caches to ensure correctness for some operations [ABK+15, NFG+13].

Given this great variety of differentiated guarantees, we surmise that applications can benefit from mixing consistency models. The notable downside of this approach is that application complexity increases [LLC+14]. Developers must orchestrate different storage APIs and consider the interactions between these protocols [ABK+15, BFF+15, WFZ+11]. Our work subsumes results in this area. We propose to hide different schemes for managing consistency under a common interface, Correctables, which can abstract over a varying combination of storage tiers and reduce application complexity. In addition, we introduce the notion of *incremental consistency guarantees* (ICG), i.e., progressive refinement of the result of a *single* operation.

6.2.2 ICG: Incremental Consistency Guarantees

Applications which use strong consistency—either exclusively or for a few operations—do so to avoid anomalous behavior which is latent in weaker models. Interestingly, recent work reveals that this anomalous behavior is rare in practice [BVF+12, LVA+15]. There are applications, however, which cannot afford to expose even those rare anomalies.

For instance, consider a system storing user passwords, and say it has 1% chance of exposing an inconsistent password. If such a system demands correctness—as it should—then it is forced to pay the price for strong consistency on *every* access, even though this is not necessary in 99% of cases. We propose ICG to help applications avert this dilemma, and pay for correctness only when inconsistencies actually occur.

With ICG, an application can obtain both weakly consistent (called *preliminary*) and strongly consistent (called *final*) results of an operation, one by one, as these become available. While waiting for the final result, the application can speculatively perform further processing based on the preliminary—which is correct on expectation. Following our earlier example, this would help hide the latency of strong consistency for 99% of accesses.

The full latency of strong consistency is only exposed in case of misspeculation, when the preliminary and final values diverge because the preliminary returned inconsistent data [WCN+09]. These are the 1% cases where strong consistency is needed anyway. Speculation through ICG can lessen the most prominent argument against strong consistency, namely its performance penalty. With ICG we pay the latency cost of strong consistency only when necessary, regardless of how often this is the case.

Speculation is a well-known technique for improving performance. Traditionally, the effects

Category	Synopsis	Applications and use cases
Weak Consistency	Use the **weakest, but fastest** consistency model, e.g., by using partial quorums, or going to the closest replica or cache. No benefit from ICG.	Computation on static (BLOBs) content, e.g., thumbnail generator for images and videos, accessing cold data, fraud analysis, disconnected operations in mobile applications, etc.
Strong Consistency	Use the **strongest** available model, e.g., by going to the primary replica. Applications require correct results.	Infrastructure services (e.g., load-balancing, session stores, configuration and membership management services), stock tickers, trading applications, etc.
Incremental Consistency Guarantees (ICG)	Use multiple, **incremental** models. Applications benefit from weakly consistent values (e.g., by speculating or exposing them), but prefer correct results.	E-mail, calendar, social network timeline, grocery list, flight search aggregation, online shopping, news reading, browsing, backup, collaborative editing, authentication and authorization, advertising, online wallets, etc.

Table 6.1 – Different types of applications building on top of replicated objects. Many of these applications can benefit from ICG. There are also applications, such as in the first two categories (weak consistency and strong consistency), which require a single consistency model, inheriting all its advantages as well as drawbacks.

of speculation in a system remain hidden from higher-level applications until the speculation confirms, since the effects can lead to irrevocable actions in the applications [KWQ+12, MEHL10, NCF05, WCN+09]. Alternatively, it has been shown that leaking speculative effects to higher layers can be beneficial, especially in user-facing applications, where the effects can be undone or the application can compensate in case of misspeculation [HC09, LDR08, LCC+15, PKFF14]. We propose to use eventual consistency as a basis for doing speculative work, as a novel approach for improving performance in replicated systems. Also, more generally, we allow the application itself (which knows best), to decide on the speculation boundary [WCF11]—whether to externalize effects of speculation, and later to undo or compensate these effects, or whether to isolate users from speculative state.

Besides speculation, ICG is useful in other cases as well. For instance, applications can choose dynamically whether to settle with a preliminary value and forsake the final value altogether. This is a way to obtain application-specific optimizations, e.g., to enforce tight latency SLAs. Alternatively, we can *expose* the preliminary response to users and revise it later when the final response arrives. This strategy is akin to compensating in case of misspeculation, as mentioned earlier.

Clearly, not all applications are amenable to exploiting ICG. In Table 6.1 we give a high-level account on three categories of applications, as follows:

1. Applications which have no additional benefit from strong consistency or ICG;
2. Application which require correct results but are not amenable to speculation; and
3. Applications that can obtain performance without sacrificing correctness via ICG.

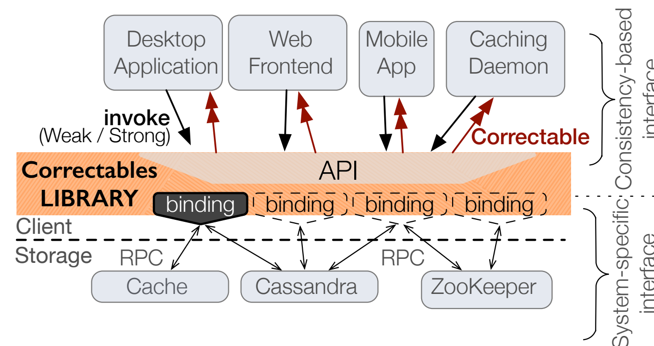


Figure 6.2 – High-level view of Correctables, as an interface to the underlying storage.

6.2.3 Client-side Handling of ICG

To program with ICG, applications need to wait asynchronously for multiple replies to an operation (where each reply encapsulates a different guarantee on the result) while doing useful work, i.e., speculate. To the best of our knowledge, no abstraction fulfills these criteria. To minimize the effort of programming with ICG, we draw inspiration from *Promises*, seminal work on handling asynchronous remote procedure calls in distributed systems [LS88].

A Promise is a placeholder for a value that will become available asynchronously in the future. Given the urgency to handle intricate parallelism and augmenting complexity in applications, it is not surprising that Promises are becoming standard in many languages [Ins19, Gua19, Fug15, Eri13]. We extend the binary interface of Promises (a value either present or absent) to obtain a multi-level abstraction, which incrementally builds up to a final, correct result.

The Observable interface from reactive programming can be seen as a similar generalization of Promises. Observables abstract over asynchronous data streams of arbitrary type and size [Mei12]. Our goal with Correctables, in contrast, is to grant developers access to consistency guarantees on replicated objects in a simple manner. The ProgressivePromise interface in Netty [Net19b] also generalizes Promises. While it can indicate progress of an operation, a ProgressivePromise does not expose preliminary results of this operation.

6.3 Correctables

We now discuss the Correctables interface for programming and speculating with replicated data. Applications use this interface as a library, depicted in Figure 6.2. At the top of this library sits the application-facing API. The library is connected to the storage stack using a storage binding—a module encapsulating all storage-system specific interfaces and protocols. Correctables fulfill two critical functions: (i) translate API calls into storage-specific requests via a binding, and (ii) orchestrate responses from the binding and deliver them—in an incremental way—to the application, using *Correctable* objects. Each call to an API method returns a Correctable which represents the progressively improving result (i.e., a result with ICG).

6.3.1 From Promises to Correctables

As mentioned earlier, Correctables descend from Promises. To model an asynchronous task, a Promise starts in the *blocked* state and transitions to *ready* when the task completes, triggering any callback associated with this state [LS88]. Promises help with asynchrony, but not incrementality. To convey incrementality, a Correctables starts in the *updating* state, where it remains until the final result becomes available or an error occurs (see Figure 6.3). When this happens, the Correctable *closes* with that result (or error), transitioning to the *final* (or *error*) state. Upon each state transition, the corresponding callback triggers. Preliminary results trigger a same-state transition (from *updating* to *updating*). A Correctable can have callbacks associated with each of its three states. To attach these callbacks, we provide the `setCallbacks` method; together with `speculate`, these two form the two central methods of a Correctables, which we examine more closely in §6.4.

6.3.2 Decoupling Semantics from Implementation

The Correctables abstraction decouples applications from storage specifics by adopting a thin, consistency-based interface, centered around *consistency levels*. This enables developers—who naturally reason in terms of consistency rather than protocol specifics—to obtain simple and portable implementations. With Correctables, applications can transparently switch storage stacks, as long as these stacks support compatible consistency models.

Our API consists of three methods:

1. `invokeWeak` (*operation*),
2. `invokeStrong` (*operation*), and
3. `invoke` (*operation*[, *levels*]).

The first two allow developers to select either weak or strong consistency for a given *operation*. The returned Correctable never transitions from *updating* to *updating* state and only closes with a final value (or error). These two methods follow the traditional practice of providing a single result which lies at one extreme of the consistency/performance tradeoff.

The third method provides ICG, allowing developers to operate on this tradeoff at run-time, which makes it especially relevant for applications in the above-mentioned gray area. Instead

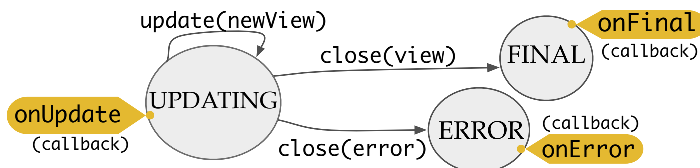


Figure 6.3 – The three states, transitions, and callbacks associated with a Correctable object.

of a single result (as is the case with the two former methods), `invoke` provides incremental updates on the operation result. Optionally, `invoke` accepts as argument the set of consistency levels which the result should—one after the other—satisfy. If this argument is absent, `invoke` provides all available levels. This argument allows some optimizations, e.g., if an application only requires a subset of the available consistency levels, this parameter informs a binding to avoid using the extraneous levels; we omit further discussion of this argument due to space constraints. The available consistency levels depend on the underlying storage system and binding, which we discuss in more detail in §6.5.

In the next section, we show how to program with Correctables through several representative use-cases. In code snippets we adopt a Python-inspired pseudocode for readability sake. For brevity we leave aside error handling, timeouts, or other features inherited from modern Promises, such as aggregation or monadic-style chaining [Fug15, Eri13, LS88].

6.4 Correctables in Action

This section presents examples of how Correctables can be useful on two main fronts. (1) Decoupling applications from their storage stacks by providing an abstraction based on consistency levels. (2) Improving application performance by means of ICG, e.g., via speculation or exploiting application-specific semantics.

6.4.1 Decoupling Applications from Storage

We first discuss a simple case of decoupling, where we illustrate the use the first two functions in our API, namely `invokeWeak` and `invokeStrong`. As discussed in §6.2, many applications differentiate between weak and strong consistency to balance correctness with performance. In practice, applications often resort to ad-hoc techniques such as cache-bypassing to achieve this, which complicates code and leads to errors [ABK+15, Eri13]. Listing 6.1 shows code from Reddit [Red16], a popular bulletin-board system and a prime example of such code. Developers have to explicitly handle cache access (lines 6 and 9), make choices based on presence of items in the cache (line 7), manually bypass the cache (line 8) under specific conditions, and write duplicate code (line 11).

Instead of explicit cache-bypassing, we can employ `invokeWeak` and `invokeStrong` to substantially simplify the code by replacing ad-hoc abstractions like `user_messages` and `user_messages_nocache`, as Listing 6.2 shows.

Furthermore, we can replace other near-identical functions for differentiated guarantees, eliminating duplicate logic.[1] Cache-coherence and bypassing is completely handled by the storage-specific binding. This reduces both programmer effort and application complexity.

[1] Similar pairs of ad-hoc functions exist in Reddit for accessing other objects. Perhaps accidentally, these other functions contain comments referring to `user_messages` instead of their specific objects. We interpret this as a strong indication of "copy-pasting" code, which Correctables would help prevent.

```
1 from pylons import app_globals as g # cache access
2 from r2.lib.db import queries       # backend access

4 def user_messages(user, update = False):
5   key = messages_key(user._id)
6   trees = g.permacache.get(key)
7   if not trees or update:
8       trees = user_messages_nocache(user)
9       g.permacache.set(key, trees) # cache coherence
10  return trees
11 def user_messages_nocache(user):
12  # Just like user_messages, but avoiding the cache...
```

Listing 6.1 – Different consistency guarantees in Reddit [Red16], as an example of tight coupling between applications and storage. Developers must manually handle the cache and the backend.

```
1 def user_messages(user, strong = False):
2   key = messages_key(user._id)
3   # coherence handled by invoke* functions in bindings
4   if strong: return invokeStrong(get(key))
5        else: return invokeWeak(get(key))
```

Listing 6.2 – Reddit code rewritten using Correctables.

The third method in our library is `invoke`. Correctables are crucial for this method, since it captures ICG. `invoke` allows applications to speculate on preliminary values (hiding the latency of strong consistency), or exploit application-specific semantics, as we show next.

6.4.2 Speculating with Correctables

Many applications are amenable to speculating on preliminary values to reap performance benefits. To understand how to achieve this, we consider any non-trivial operation in a distributed application which involves reading data from storage. Using `invoke` to access the storage, applications can perform speculation on the preliminary value. If this preliminary value is confirmed by the final value, then speculation was correct, reducing overall latency [WCN+09]. Examples where speculation applies include password checking or thumbnail generation (as mentioned in [TPK+13]), as well as operations for airline seat reservation [YV00], or web shopping [KHAK09].

Listing 6.3 depicts how this is performed in practice with Correctables. Even though such speculation can be orchestrated directly by using the `onUpdate` and `onFinal` callbacks of a Correctable object, we provide a convenience method called `speculate` that captures the speculation pattern (L2). It takes a speculation function as an argument, applying it to every

```
1 invoke(read(...))
2   .speculate(speculationFunc[, abortFunc])
3   .setCallbacks(onFinal = (res) => deliver(res))
```

Listing 6.3 – Generic speculation with Correctables. The square brackets indicate that `abortFunc` function is optional.

```
1 def fetchAdsByUserId(uid):
2   invoke(getPersonalizedAdsRefs(uid))
3     .speculate(getAds) # fetch & post−process ads
4     .setCallbacks(onFinal = (ads) => deliver(ads))
```

Listing 6.4 – Example of applying speculation in an advertising system to hide latency of strong consistency.

new view delivered by the underlying Correctable if this view differs from the previous one. The `speculate` method returns a new Correctable object which closes with the return value of the user-provided speculation function. If the final view matches a preliminary one (which is the common case), the new Correctable can close immediately when the final view becomes available, confirming the speculation. Otherwise, it closes only after the speculation function is (automatically) re-executed with correct input. In the latter case, an optional abort function is executed, undoing potential side-effects of the preceding speculation. Next, we discuss an ad serving system as an example application that can benefit from such speculation.

Advertising System. Typically, ads are personalized to user interests. These interests fluctuate frequently, and so ads change accordingly [Kor10]. Given their revenue-based nature, advertising systems have conflicting requirements, as they aim to reconcile consistency (freshness of ads) with performance (latency) [CRS+08, CDE+13]. We thus find that they correspond to our notion of gray area, and are a suitable speculation use-case.

Listing 6.4 shows how we can use ICG while fetching ads. First, we obtain a list of *references* to personalized ads using the `invoke` method (L2). This method returns both a preliminary view (with weak guarantees) and a final (fresh) view. Using the references in the preliminary view, we fetch the actual ads content and media, and do any post-processing, such as localization or personalization (L3). If the final view corresponds to the preliminary, then speculation was correct, and we can deliver (L4) the ads fast; otherwise, `getAds` re-executes on the final view, and we deliver the result later. This application is our first experimental case-study (§6.6.3).

The pattern of fetching objects based on their references—which themselves need to be fetched first—is widespread. It appears in many applications, such as reading the latest news, the most recent transactions, the latest updates in a social network, an inventory, the most pressing items in a to-do list or calendar, and so on. In all these cases, the application needs

to chase a pointer (reference) to the latest data, while weak consistency can reveal stale values, which is undesirable. We avoid stale data by reading the references with `invoke`, and we mask the latency of the final value by speculatively fetching objects based on the preliminary reference.

6.4.3 Exploiting Application Semantics

Applications can exploit their specific semantics to leverage the preliminary and the final values of `invoke`. For instance, consider the web auction system mentioned by Kraska et al. [KHAK09], where strong consistency is critical in the last moments of a bid, but is not particularly helpful in the days before the bid ends, when contention is very low and anomalous behavior is unlikely. Another example is selling items from a predefined stock of such items. If a preliminary response suggests that the stock is still big, it is safe to proceed with a purchase. Otherwise, if the stock is almost empty, it would be better to wait for the arrival of the final response. This is the case, for instance, for a system selling tickets to an event, which we describe next.

Selling Tickets for Events. For this application system, we depart from the popular key-value data type. First, as we want to avoid overselling, we need a stronger abstraction to serialize access to the ticket stock. Simple read/write objects (without transactional support) are fundamentally insufficient [Her91]. Second, we want to demonstrate the applicability of ICG to other data types. We thus model the ticket stock using a queue, which is a simple object, yet powerful enough to avoid overselling.

Event organizers enqueue tickets and retailers dequeue them. This data type allows us to serialize access to the shared ticket stock [AMS+07, KHAK09]. We assume, however, that tickets bear no specific ordering (i.e., there is no seating). Clients are interested in purchasing *some* ticket, and it is irrelevant which exact element of the queue is dequeued. We can thus resort to weak consistency most of the time, and use strong consistency sparingly. We consider a weakly consistent result of an operation to be the outcome of simulating that operation on the local state of a single replica (see §6.5.2).

Listing 6.5 shows how we can selectively use strong consistency in this case, based on the estimated stock size. For each purchase, retailers use `invoke` with the dequeue operation. This yields a quick preliminary response, by peeking at the queue tail on the closest replica of the queue. If the preliminary value indicates that there are many tickets left (e.g., via a ticket sequence number, denoting the ticket's position in the queue), which is the common case, the purchase can succeed without synchronous coordination on dequeue, which completes in the background. This reduces the latency of most purchase operations. As the queue drains, e.g. below a predefined threshold of 20 tickets, retailers start waiting for the final results, which gives atomic semantics on dequeuing, but incurs higher latency. This system represents our second experimental case study (§6.6.3).

```
1 def purchaseTicket(eventID):
2   done = false
3   invoke(dequeue(eventID)).setCallbacks(
4     onUpdate = (weakResult) =>
5       if weakResult.ticketNr > THRESHOLD:
6         done = true # many tickets left, so we can buy
7         confirmPurchase()
8     onFinal = (strongResult) =>
9       if not done:
10        if strongResult is not null:
11          confirmPurchase() # we managed to get a ticket
12        else: display("Sold out. Sorry!"))
```

Listing 6.5 – Dynamic selection of consistency guarantees in a ticket selling system. If there are many tickets in the stock, we can safely use weak consistency.

```
1 invoke(getLatestNews()).setCallbacks(
2     onUpdate = (items) => refreshDisplay(items))
```

Listing 6.6 – Progressive display of news items using Correctables. The `refreshDisplay` function triggers with every update on the news items.

6.4.4 Exposing Data Incrementally

In some cases, it is beneficial to expose even incorrect (stale) data to the user if this data arrives fast, and amend the output as more fresh data becomes available. Indeed, a quick approximate result is sometimes better than an overdue reply [DHJ+07, TPK+13]. Many applications update their output as better results become available. A notable example is flight search aggregation services [Sky], or generally, applications which exhibit high responsiveness by leaking to the user intermediary views on an ongoing operation [LDR08, LCC+15], e.g., previews to a video or shipment tracking. We can assist the development of this type of applications, as we describe next.

Smartphone News Reader. Consider a smartphone news reader application for a news service replicated with a primary-backup scheme [TPK+13]. Additionally, recently seen news items are stored in a local phone cache. With ICG provided by Correctables, the application can be oblivious to storage details. It can use a single logical storage access to fetch the latest news items, as Listing 6.6 shows. The binding would translate this logical access to three actual requests: one to the local cache, resolving almost immediately, one to the closest backup replica, providing a fresher view, and one to a more distant primary (i.e., leader) replica, taking the longest to return but providing the most up-to-date news stories.

6.4.5 Discussion: Applicability of ICG

In a majority of use-cases, we observe that two views suffice. Correctables, however, support arbitrarily many views. Note that this does not add any complexity to the interface and can be useful, as the news reader application shows.

There are other examples of applications which can benefit from multiple views. A notable use-case are blockchain-based applications (e.g., Bitcoin [Nak08]), where Correctables can track transaction confirmations as they accumulate and eventually the transaction becomes an irrevocable part of the blockchain, i.e., strongly-consistent with high probability. This is a use-case we also implemented, but omit. In larger quorum systems (e.g., BFT), Correctables can represent the majority vote as it settles. Search or recommender systems, likewise, can benefit from exposing multiple intermediary results in subsequent updates.

Intuitively, multiple preliminary views are helpful for applications requiring live updates. On the one hand, several preliminary values would make the application more interactive and offer users a finer sense of progress. This is especially important when the final result has high latency (Bitcoin transactions take tens of minutes). On the other hand, as the replicated system delivers more preliminary views for an operation, less operations can be sustained and overall throughput drops. Thus, applications which build on ICG with multiple incremental views observe a tradeoff between interactivity and throughput. This tradeoff can be observed even when the system delivers only two views (§6.6.2).

In order to be practical, the cost of generating and exploiting the preliminary values of ICG must not outweigh their benefits. The cost of generating ICG is captured in the tradeoff we highlighted above; the cost of exploiting ICG is highly application-dependent. If used for speculation, the utility of 2+ views depends on how expensive it is to re-do the speculative work upon misspeculation. This can range from negligible (simply display preliminary views) to potentially very expensive (prefetch bulky data). Additionally, the utility also depends on how often misspeculation actually occurs. This depends on the workload characteristics: workloads with higher write ratios elicit higher rates of inconsistencies, and thus more misspeculations (§6.6.2–Divergence).

There are also cases when using ICG is not an option. This is either due to the underlying storage providing a unique consistency model and lacking caches, or due to application semantics, which can render ICG unnecessary—we give examples of this in the first two rows of Table 6.1. Correctables, however, are beneficial beyond ICG. This abstraction can hide the complexity of dealing with storage-specific protocols, e.g., quorum-size selection. The application code thus becomes portable across different storage systems.

6.5 Bindings

Our library handles all the instrumentation around Correctables objects. This includes creation, state transitions, callbacks, and the API inherited from Promises [Fug15, Eri13]. Bindings are storage-specific modules which the library uses to communicate with the storage. These modules encapsulate everything that is storage system specific, and thus draw the separating line between consistency models—which Correctables expose—and implementations of these models. In this section, we describe the binding API, and show how bindings can facilitate efficient implementation of ICG with server-side support.

6.5.1 Binding API

An instance of our library always uses one specific binding. A binding establishes: (1) the concrete configuration of the underlying storage stack (e.g., Memcache on top of Cassandra) together with (2) the *consistency levels* offered by this stack, and (3) the implementation of any storage specific protocol (e.g., for coherence, choosing quorums). This allows the library to act as a client to the storage stack.

When an application calls an API method (§6.3.2), the library immediately returns a Correctables. In the background, we use the *binding API* to access the underlying storage. The binding forwards responses from the storage through an upcall to the library. The library then updates (or closes) the associated Correctables, executing the corresponding callback function.

The binding API exposes two methods to the library. First, `consistencyLevels()` advertises to the library the supported consistency levels. It simply returns a list of supported consistency levels, ordered from weakest to strongest. In most implementations, this will probably be a one-liner returning a statically defined list. The second function is `submitOperation(op, consLevels, callback)`. The library uses this function to execute operation `op` on the underlying storage, with `consLevels` specifying the requested consistency levels. The `callback` activates whenever a new view of the result is available. The binding has to implement the protocol for executing `op` and invoke `callback` once for each requested consistency level.

Listing 6.7 shows the implementation of a simple binding for a primary-backup storage, supporting two consistency levels. A more sophisticated binding could access the backup and primary in parallel, or could provide more than two consistency levels. We designed the binding API to be as simple as possible; contributors or developers wishing to support a particular store must implement this API when adding new bindings. We currently provide bindings to Cassandra and ZooKeeper.

6.5.2 Efficiency and Server-side Support

On a first glance, ICG might seem to evoke large bandwidth and computation overheads. Indeed, if the `invoke` method comprises multiple independent single-consistency requests,

```
1 def consistencyLevels():
2   return [WEAK, STRONG]

4 def submitOperation(operation, consLevels, callback):
5   if WEAK in consLevels:
6     backupResult = queryClosestBackup(operation)
7     callback(backupResult, WEAK)
8   if STRONG in consLevels:
9     primaryResult = queryPrimary(operation)
10    callback(primaryResult, STRONG)
```

Listing 6.7 – Simple binding to a storage system with primary-backup replication.

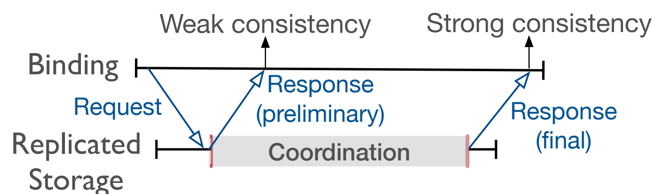


Figure 6.4 – Simple server support for efficient ICG. The storage system sends a preliminary response before coordinating. Note that for a *single* request, the storage provides *two* responses.

then storage servers will partly redo their own work. Also, as the weakly and strongly consistent values often coincide, multiple responses are frequently redundant. Such overheads would reduce the practicality of ICG.

With server-side support, however, we can minimize these overheads. For instance, we can send a *single* request to obtain all the incremental views on a replicated object. An effective way to do this is to hook into the coordination mechanism of consistency protocols. This mechanism is the core of such protocols, and the provided consistency model and latency depend on the type of coordination. For example, asynchronous coordination is off the critical path of client requests and ensures eventually consistent results with low-latency [DHJ+07]. Coordination through an agreement protocol as Paxos [Lam98] yields linearizability [HW90], but at a higher latency price.

Our basic insight is that we can get a good guess of the result already before coordinating, based on a replica's local state. In fact, this same state is being exposed when asynchronous coordination is employed, and as we already mentioned, this state is consistent on expectation. The replica can leak a preliminary response—with weak guarantees—to the client prior to coordination (Figure 6.4). Moreover, we can reduce bandwidth overhead by skipping the final response if it is the same as the preliminary: a small *confirmation* message suffices, to indicate that the preliminary response was correct. Indeed, with such an optimization, ICG has minor bandwidth overhead (§6.6.2).

An additional benefit from this approach compared to sending two independent requests is that it prevents certain types of unexpected outcomes. For instance, strong consistency might be more stale than weak consistency if responses to two independent requests were reordered by the WAN [TPK+13]. Using this approach, we modify two popular systems—Cassandra and ZooKeeper—to provide efficient support for ICG. Other techniques (e.g., master leases [CGR07]) or replication schemes (e.g., primary-backup) can provide final views fast, skipping the preliminary altogether.

Cassandra. Cassandra uses a quorum-gathering protocol for coordination [Gif79]. In our modified version of Cassandra—called Correctable Cassandra (CC)—the coordinating node sends a preliminary view after obtaining the *first* result from any replica. This view has low latency, obtained either locally (if the coordinator is itself a replica) or from the closest replica. Our binding to CC supports two consistency levels, *weak* (involving one replica) and *strong* (involving two or more). To minimize bandwidth overhead of `invoke`, CC uses the confirmation messages optimization we mentioned earlier.

ZooKeeper. To demonstrate the versatility of Correctables, we consider a different data type, namely replicated queues, which ZooKeeper can easily model [Zoo19]. Our binding supports operations `enqueue` and `dequeue`, with weak and strong consistency semantics, accessible via `invokeWeak` and `invokeStrong`, respectively; `invoke` supplies both consistency models incrementally.

The vanilla ZooKeeper implementation (ZK) has strong consistency [HKJR10]. For efficient ICG, we implement Correctable ZooKeeper (CZK) by adding a fast path to ZK: a replica first simulates the operation on its local state, returning the preliminary (weak) result. After coordination (via the ZAB protocol [JRS11]), this replica applies the operation and returns the strong response.

Causal Consistency and Caching. We also implement a binding to abstract over a causally consistent store complemented by a client-side cache. The `invoke` function reveals two views: one from cache (very fast, possibly stale), and another from the causally consistent store. This binding ensures write-through cache coherence, allows cache-bypassing (`invokeStrong`) or direct cache access (`invokeWeak`), e.g., in case of disconnected operations for mobile applications [PAA+15]. Given the space constraints we focus on the two other bindings.

6.6 Evaluation

Our evaluation focuses on quantifying the benefits of ICG. Before diving into it, it is important to note that any potential benefit of ICG is capped by performance gaps among consistency models. Briefly, if strong consistency has the same performance as weaker models (or the

difference is negligible) then applications can directly use the stronger model. This is, however, rarely the case. In practice, there can be sizable differences—up to orders of magnitude—across models [BDF+13, TPK+13].

We first describe our evaluation methodology, and then show that such optimization potential indeed exists. We do so by looking at the performance gaps between weak and strong consistency in quorum-based (Cassandra) and consensus-based (ZooKeeper) systems. We then quantify the performance gain of using ICG in three case studies: a Twissandra-based microblogging service [Twi19], an ad serving system, and a ticket selling application.

6.6.1 Methodology

We run all experiments on Amazon's EC2 with m4.large instances and a replication factor of 3, with replicas distributed in Frankfurt (FRK), Ireland (IRL), and N. Virginia (VRG). Unless stated otherwise, to obtain WAN conditions, the client is in IRL and uses the replica in FRK; note that colocating the client with its contact server (i.e., both in IRL) would play to our advantage, as it would reduce the latency of preliminary responses and allow a bigger performance gap. We also experiment with various other client locations in some experiments.

For Cassandra experiments, we compare the baseline Cassandra v2.1.10 (labeled C), with our modified Correctable Cassandra (CC). We use superscript notation to indicate the specific quorum size for an execution, e.g., C^1 denotes a client reading from Cassandra with a read quorum $R = 1$ (i.e., involving 1 out of 3 replicas). For the ZooKeeper queue, we compare our modified Correctable ZooKeeper (CZK) against vanilla ZooKeeper (ZK), v3.4.8. The cumulative implementation effort associated with CC and CZK, including three case studies, is modest, at roughly $3k$ LOC Java code.

6.6.2 Potential for Exploiting ICG

To determine the potential of ICG, we examine their behavior in practice. Studies show that large load on a system and high inter-replica latencies give rise to large performance gaps among consistency models [BDF+13, TPK+13]. To the best of our knowledge, however, there are no studies which consider a combination of incremental consistency models in a single operation. We first investigate this behavior in Cassandra and then in ZooKeeper.

Potential for Exploiting ICG in Cassandra

Cassandra can offer us insights into the basic behavior of ICG in a quorum system. As explained in §6.5, CC offers two consistency models: weak, which yields the *preliminary* view ($R = 1$), and strong, giving the *final* view ($R = 2$ or $R = 3$, depending on the requested quorum size). For write operations, we set $W = 1$. We use microbenchmarks and YCSB [CST+10] to measure single-request latency and performance under load, respectively. For each CC exper-

iment, we run three 60-second trials and elide from the results the first and last 15 seconds. We report on the average and 99th percentile latency, omitting error bars if negligible.

Single-request Latency. We use a microbenchmark consisting of read-only operations on objects of $100B$. We are interested in the performance gap between preliminary and final views as provided by ICG, and we contrast these with their vanilla counterparts. We thus compare CC^2 ($R \in \{1,2\}$) and CC^3 ($R \in \{1,3\}$) with C^1 ($R = 1$), C^2 ($R = 2$), and C^3 ($R = 3$). For CC, R has two values: the read quorum size for the preliminary (weak) and for the final (strong) replies, respectively.

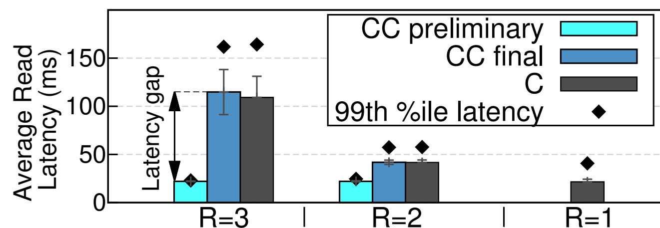


Figure 6.5 – Single-request latencies in Cassandra for different quorum configurations. A bigger latency gap means a larger time window available for speculation.

Figure 6.5 shows the results for all these configurations, grouped by their read quorum size. The average latency of preliminary views—whether it is for CC^2 or CC^3—follows closely the latency of C^1, which coincides with the $20ms$ RTT between the client and the coordinator. Preliminary views reflect the local state on the replica in FRK, having the same consistency as C^1. Final views of CC^2 and CC^3 follow the trend of the requested quorum size and reflect the behavior of C^2 and C^3 respectively.

The performance gap between the preliminary and final view for CC^2 is $20ms$. The coordinator (FRK) is gathering a quorum of two: itself and the closest replica (IRL). The gap indeed corresponds to the RTT between these two regions. For CC^3, the gap is much larger: up to $140ms$ for the 99th percentile, due to the larger distance to reach the third replica (VRG). By speculating on the preliminary views, applications can hide up to $20ms$ (or $140ms$) of the latency for stronger consistency. In practice, such differences already impact revenue, as users are highly-sensitive to latency fluctuations [DHJ+07, Ham09].

Performance Under Load. We also study the performance gap using YCSB workloads A (50:50 read/write ratio), B (95:5 read/write ratio), and C (read-only) [CST+10]. To stress the systems and obtain WAN conditions, we deploy 3 clients, one per region, with each client connecting to a remote replica. For brevity, we only report on the results for the client in IRL and $R = \{1,2\}$. Figure 6.6 presents the average latency as a function of throughput. We plot the evolution of both the preliminary and final views individually.

We observe that CC trades in some throughput due to the load generated on the coordinator,

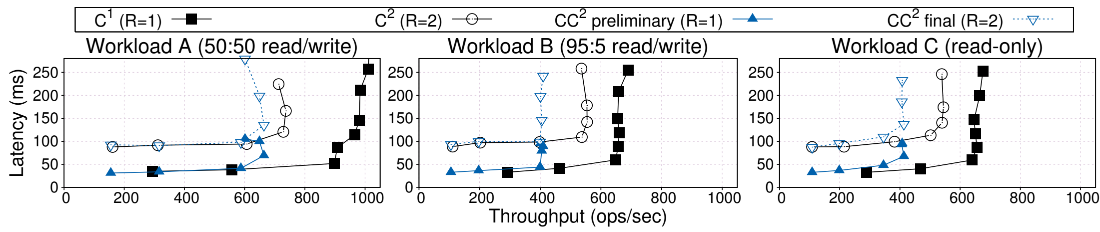


Figure 6.6 – Performance of Correctable Cassandra (CC) compared to baseline Cassandra (C). Note that the measurements for CC^2 have two results, one for the preliminary view and another for final. These two have the same throughput but different latencies.

which handles ICG. We observe this behavior in all three workloads. This is to be expected, considering the modifications necessary to implement preliminary replies (§6.5.2). Briefly, we add another step to every read operation that uses quorums larger than one. This step, called *preliminary flushing*, occurs at any coordinator replica serving read operations as soon as that replica finishes reading the requested data from its local storage—and prior to gathering a quorum from other replicas. This step generates additional load on the coordinator replica, explaining the throughput drop of CC^2 compared to baselines. Related work on replicated state machines (RSM) suggests an optimization [WCN+09] which resembles our flushing technique. Perhaps unsurprisingly, the optimized RSM exhibits a similar throughput drop [WCN+09, §6.2] as we notice in these experiments.

The latency gap between preliminary and final views is the same as the one we observe in the microbenchmarks. To conclude, our results confirm that the performance gaps while using ICG are noticeable, and hence there is room for hiding latency.

Divergence. To obtain more insight about the behavior of ICG, we use CC and the YCSB benchmark to measure how often preliminary values diverge from final results. We achieve this by using `invoke` and comparing the preliminary view to the final one. We run this experiment with a small dataset of $1K$ objects. We aim at obtaining the conditions of a highly-loaded system where clients are mostly interested in a small (popular) part of the dataset.

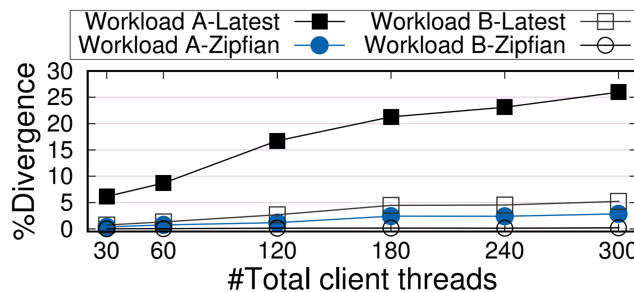


Figure 6.7 – Divergence of preliminary from final (correct) views in Correctable Cassandra with various YCSB configurations.

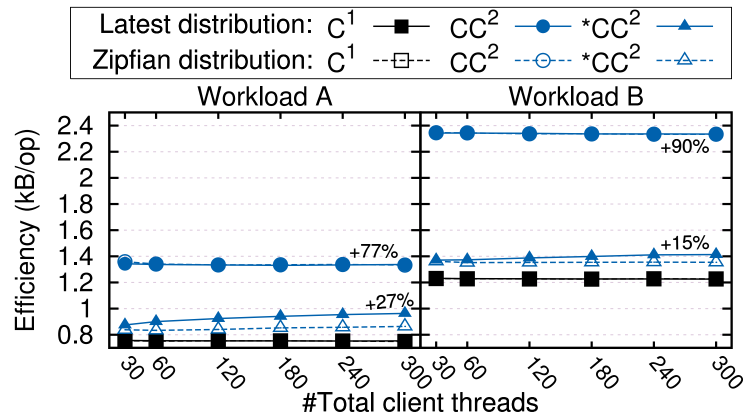


Figure 6.8 – Efficiency (quantified by bandwidth overhead) of the ICG implementation in Correctable Cassandra.

Figure 6.7 shows our result for a mix of representative YCSB workloads (A and B) and access patterns (Zipfian and Latest) with default settings. Notably, workload A (50:50 read-/write) under Latest distribution (read activity skewed towards recently updated items) exhibits high divergence, up to 25%. Under such conditions, using $R = 1$ would yield many stale results. Indeed, some applications with high write ratios, e.g., notification or session stores [CST+10, HK11], tend to use $R = 2$, even though this forces *all* read operations to pay the latency price [BVF+12].

In fact, even if less than 1% of accessed objects are inconsistent, these are typically the most popular ("linchpin" [ABK+15, NFG+13]) objects, being both read- and write-intensive. Such anomalies have a disproportionate effect at application-level, since they reflect in many more than 1% application-level operations. Applications with high update ratios as modeled by workload A, e.g., social networks [CRS+08], can thus benefit from exploiting ICG to avoid anomalies.

Bandwidth Overhead. In addition to the throughput drop mentioned above, client-replica bandwidth is the next relevant metric which ICG can impact. Yet, optimizations can cut the cost of this feature (§6.5.2). We implement such an optimization in CC, whereby a final view contains only a small confirmation—instead of the full response—if it coincides with the preliminary view. We note that in all experiments thus far we did not rely on this optimization, which makes our comparisons with Cassandra conservative.

To obtain a worst-case characterization of the costs of ICG, we consider the scenario where divergence can be maximal, as this will lessen the amount of bandwidth we can save with our optimization. Hence, we consider the exact conditions we use in the divergence benchmark, where we discovered that divergence can rise up to 25%. In this experiment, we measure the average data transferred (KB) per operation. We contrast three scenarios. First, as baseline, we use C^1, where clients request a single consistency version using weak reads. The

other two systems are CC^2 (without optimization) and $^*CC^2$ (optimized to reduce bandwidth overhead).

Figure 6.8 shows our results. As expected, if divergence is very high—notably in workload A—then many preliminary results are incorrect. This means that final views cannot be replaced by confirmations, increasing the data cost by up to 27%. Without any optimization, this would drive the cost up by 77%. Workload B has a smaller write ratio (5%), so a lower divergence and more optimization potential: we can reduce the overhead from 90% down to 15% (since most final views are confirmations).

Our experiments prove that ICG have a modest cost in terms of data usage. This cost can be further reduced through additional techniques (§6.5.2). We remark that our choice of baseline, C^1, is conservative, because CC^2 offers better guarantees than C^1. A different baseline would be a system where clients *send two requests*—one for $R = 1$ and one for $R = 2$—and *receive two replies*. While such a baseline offers the same properties as CC^2, it would involve bigger data consumption, putting our system at an advantage.

Potential for Exploiting ICG in ZooKeeper

Latency Gaps. We also measure performance gaps in ZooKeeper queues for various locations of the leader and the replica which the client (in IRL) connects to. We show the results for four representative configurations for adding elements to a queue (we discuss dequeuing in the context of a ticket selling system in §6.6.3). The elements are small, containing an identifier of up to $20B$ (e.g., ticket number). Figure 6.9 shows the latency gaps when we use ICG in Correctable ZooKeeper (CZK) compared to baseline ZooKeeper (ZK).

In all cases, the latency of the preliminary view (containing the name of the assigned znode) corresponds to the RTT between the client and the contacted replica. This latency ranges from $2ms$ (when client and replica are both in IRL), through $20ms$ (the RTT from IRL to FRK), up to $83ms$ (the RTT between IRL and VRG). The most appealing part of this result is perhaps

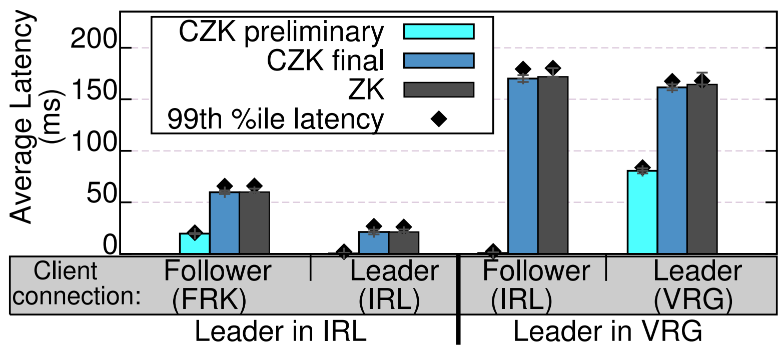


Figure 6.9 – Latency gaps between preliminary and final views on the result of dequeue operations in Correctable ZooKeeper (CZK) compared to ZooKeeper (ZK). Client is in IRL.

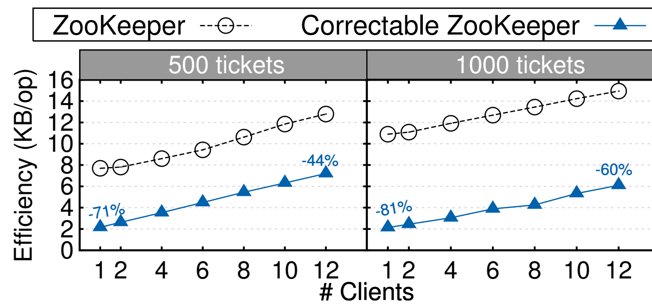


Figure 6.10 – Efficiency (bandwidth overhead) for dequeuing operation in Correctable ZooKeeper (CZK) and ZooKeeper (ZK). Overhead in CZK is independent of queue size.

the substantial gap which appears when the client and the closest follower are in IRL and the leader is distant (in VRG), in the third group of results in Figure 6.9.

Bandwidth Overhead. Storing big chunks of data is not ZooKeeper's main goal. The client-server bandwidth is usually not dominated by the payload, reducing the benefits of the confirmation optimization. For enqueuing, the bandwidth cost thus increases by roughly 50%, from 270 to 400 bytes/operation. As expected, this corresponds to one additional (preliminary) response message in addition to the original request and (final) response.

While queues are a common ZooKeeper use-case, a problem appears in standard dequeue implementations due to message size inflation [Net19a]. Specifically, clients first read the *whole queue* and then try to remove the tail element. To evade this problem in CZK, clients only read the constant-sized tail relevant for dequeuing. Figure 6.10 compares the bandwidth cost per dequeue operation in CZK and ZK for different queue sizes as we increase the number of contending threads. While the cost still increases with contention in both cases, in CZK we make it independent of queue size, which is not the case for ZK. As future work, we plan to make the dequeue cost also independent of contention using tombstones [SS05].

6.6.3 Case Studies for Exploiting ICG

Given the optimization potential explored so far, we now investigate how to exploit it in the context of three applications: the Twissandra microblogging service [Twi19], an ad serving system, and a ticket selling system. The first two build on CC and use speculation. The last application uses CZK queues.

Speculation Case Studies

For Twissandra, we are interested in `get_timeline` operation, since this is a central operation and is amenable to optimization through speculation. This operation proceeds in two-steps: (1) fetch the timeline (tweet IDs), and then (2) fetch each tweet by its ID. We re-implement this

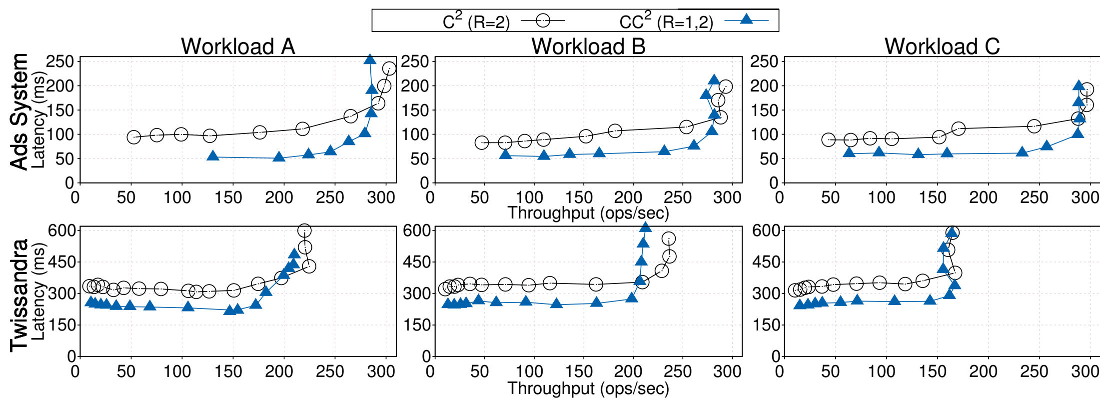


Figure 6.11 – Using speculation via ICG to improve latency in the advertising system and in Twissandra (`get_timeline` operation). Correctable Cassandra (CC) improves latency by up to 40% in exchange for a throughput drop of 6%.

function to use `invoke` on step (1) and leverage the preliminary timeline view to speculatively execute step (2) by prefetching the tweets. If the final timeline corresponds to the preliminary, then the prefetch was successful and we can reduce the total latency of the operation. In case the final timeline view is different, we fetch the tweets again based on their IDs from this final view.

Our second speculation case study is the ad serving system we describe in §6.4.2. The goal is to reduce the total latency of `fetchAdsByUserId` operation without sacrificing consistency, so we exploit ICG by speculating on preliminary values (Listing 6.4).

For both systems, we adapt their respective operations to use `invoke` ($R = \{1,2\}$) and plug them in the YCSB framework. We compare these operations using a baseline that uses only the strongly consistent result ($R = 2$), and does not leverage speculation. For Twissandra we use a corpus of $65k$ tweets [Twe19] spread over $22k$ user timelines; the ad serving system uses a dataset of $100k$ user-profiles and $230k$ ads, where each profile references between 1 and 40 random ads.

The results are in Figure 6.11. In contrast to our other experiments, we deploy Twissandra replicas in Virginia, N. California, and Oregon EC2 regions. The goal is to see how performance gains vary based on deployment scenario. The ads system uses the same configuration as before. The client is in IRL for both experiments.

We first explain the results for the ads system. As can be seen, these are consistent with our earlier findings from Cassandra experiments (Figure 6.6). We trade throughput for better latency. Prior to saturation, we can serve ads with an average latency of $60ms$. In the same conditions, the baseline achieves $100ms$ average latency (improvement by 40%). In turn, the

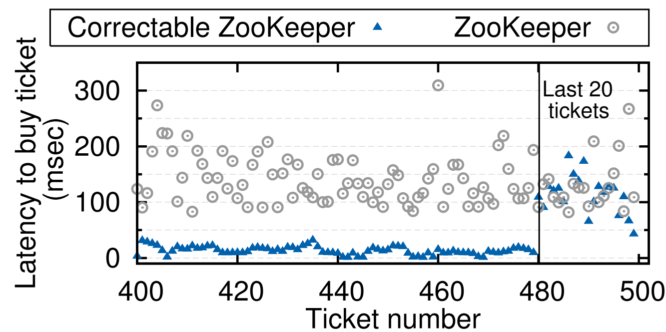


Figure 6.12 – Selling tickets with ZK and CZK. The last 20 tickets incur high latency due to strong consistency.

throughput drop is most noticeable in workload A, by $18 ops/sec$ (reduced by 6%). The smaller throughput drop compared to the raw results of Figure 6.6 is explained by the fact that each `fetchAdsByUserId` entails two storage accesses. Only the first access, however, uses ICG (to speculate). The second storage access is hidden inside `getAds` (Listing 6.4, L3); this is a read with $R = 2$, incurring no extra cost.

For Twissandra, we observe a lower throughput and higher latency, as the client is farther from the coordinator and replicas are also more distant from each other. But otherwise we draw similar conclusions. Notably, across both of these case-studies, divergence was consistently under 1%, so the applications encountered very few misspeculations.

Selling Tickets to Events

A second notable use-case of ICG is exploiting application semantics, as we discuss in the ticket selling system from §6.4.3 (see Listing 6.5). Here we exploit the fact that the position of a ticket in the queue is irrelevant. Thus, in the common case, we can rely on the preliminary value. Strong consistency (atomicity), however, becomes critical when ticket retailers are contending over the last few remaining tickets. Using ICG, we can switch dynamically between using the preliminary or the final results when the stock becomes low, to avoid overselling.

We consider 4 retailers concurrently serving (dequeuing) tickets from a fixed-size stock of 500 tickets. Retailers are co-located with a CZK follower in FRK, the leader being in IRL. We wait for the final (atomic, equivalent to ZK) response for the last 20 tickets, otherwise we use the preliminary one. This is a conservative bound; in our experiments, only the last two tickets were “revoked” by the final view on average, with a maximum of six.

Figure 6.12 shows individual ticket purchase latencies, averaged over five runs, compared to latencies with vanilla CZK. As long as there are more than 20 tickets left, we reduce the purchase latency substantially. The high variability of final view latencies is caused by contention between the retailers, which does not affect preliminary views. We experiment also with larger ticket stocks (1000), but the queue length has no practical effect on latencies. To support more

contention (more retailers) in practice, such a ticketing service can scale-out. For instance, we can shard the ticket stock and instantiate multiple replicated CZK services, each of them serving a partition of the overall stock, ensuring scalability [Bur06].

6.7 Concluding Remarks

In this chapter we presented Correctables, an abstraction for programming with replicated objects. The contribution of Correctables is twofold. First, they *decouple* an application from its underlying storage stack by drawing a clear boundary between consistency guarantees and the various methods of achieving them. This reduces developer effort and allows for simpler and more portable code.

Second, Correctables provide *incremental consistency guarantees* (ICG), which allow to compose multiple consistency levels within a single operation. With this type of guarantees we aim to fill a gap in the consistency/performance tradeoff. Namely, applications can make last-minute decisions about what consistency level to use in an operation while this operation is executing. This opens the door to new optimizations based on speculation or on concrete, application-specific semantics.

We evaluated the performance and overhead of ICG, as well as the impact of this novel type of guarantees on three practical systems: (1) a microblogging service and (2) an ad serving system, both backed by Cassandra, and (3) a ticket selling system based on ZooKeeper queues. We modified both Cassandra and ZooKeeper to support ICG with little overhead. We showed how ICG provided by Correctables bring substantial latency decrease for the price of small bandwidth overhead and throughput drop.

We believe that Correctables provide a new way to structure the interaction between applications and their storage by exploiting incrementality. This enables new designs of distributed systems that access replicated objects more efficiently.

Part IV

Supporting Efficient Token Transfers

7 Asynchronous Token Transfers

The form of a building must follow its function.
— Henry Cameron (from Ayn Rand's *The Fountainhead*)

In this chapter we focus on token-based applications. The goal is to implement a distributed system that enables the transfer of tokens (i.e., fungible resources such as digital coins, ownership documents, or payment proofs). This class of applications typically assumes a Byzantine environment where participants are mutually distrustful. The most popular use-case, by far, is to enable transfers of coins between system users, i.e., online payments [Nak08].

We propose the abstraction of an exclusive token account, or Exa. This abstraction models a container for tokens such as a bank account or wallet. The defining characteristic of an Exa object is that it has a unique owner—one of the users in the system—and this owner is exclusively allowed to spend tokens from that account. A recent result shows that such an object can be implemented asynchronously, i.e., without consensus [GKM+18, §2]. Building on this result, we introduce Astro (Asynchronous Token Transfer Protocol), a system implementing the Exa abstraction. We evaluate Astro in three scenarios (fault-free, crash-fault, and asynchronous network), showing that it outperforms a consensus-based solution.

7.1 Introduction

There is a growing body of work on implementing digital trust solutions. This means enabling users—despite their mutual distrust—to coordinate their updates on some shared application data. In other words, the addressed problem is that of achieving Byzantine fault-tolerant (BFT) replication [LSP82], covered earlier in this dissertation (§1.1).

Examples of digital trust applications include financial records management, public goods tracking, and many others [CPVK16, Kem17]. We consider a specific class of digital trust applications, which we call token-based applications. This class of applications typically allow users in the system to transfer tokens (i.e., fungible resources) between each other. Intuitively,

these applications model practical problems such as online payments [Nak08], electronic voting, land registers, a subset of the functionality of token accounts like ERC20 [VB15], or registries [But15].

As we described already (§1.1), typical solutions for implementing digital trust rely on a consensus algorithm, seeking to obtain a total order across all operations in the system. Consensus is a central problem in distributed computing, known for its notorious difficulty. Consensus has no deterministic solution in asynchronous systems, such as the wide-area network (WAN), if just a single participant can fail [FLP85]. Algorithms for solving consensus are tricky to implement correctly [AGM+17, CV17, CWA+09], and they face tough tradeoffs between efficiency, security, and scalability [Vuk15].

While solutions such as Hyperledger [ABB+18, SBV18], Bitcoin [Nak08], or Ethereum [Woo15] prove the feasibility of token-based applications, they also point out their efficiency or scalability problems [Vuk15]. The idea of token-based applications for digital trust is therefore very appealing, but challenging.

In this chapter we revisit the problem of achieving efficient token transfers, approaching this from a top-down perspective. Our starting point is a recent result by Guerraoui et al. [GKM+18] showing that consensus is not necessary for implementing token-based applications. Concretely, this result presents an asset-transfer sequential object type and a corresponding wait-free implementation, showing that this type has consensus number *one* [GKM+18, §2].

We build on this high-level idea that token transfers do not require solving consensus. Our goal is to bypass this building block, and obtain a solution that is more efficient than traditional ones based on consensus. For doing so, we propose the abstraction of an exclusive token account, or Exa. An Exa object models the account of a specific user in the system. The central characteristic of the Exa type is that it has a unique owner and only this owner may manipulate the Exa object by spending tokens (i.e., transferring) from that account.

We describe a system, called Astro (Asynchronous Token Transfer Protocol), which implements the Exa abstraction for the Byzantine environment. In Astro we do not impose a total order across all transfers in the system. Instead, we rely on relaxed ordering guarantees that can be achieved without resorting to consensus. More precisely, Astro employs a broadcast primitive with causal ordering guarantees which can be implemented asynchronously.

It has been observed that often the main bottleneck in digital trust systems is their consensus module [Hea16, SBV18, Vuk15]. Intuitively, a broadcast-based solution should outperform a consensus-based system. We put this intuition to the test, by evaluating Astro in various scenarios, including in an asynchronous network, and comparing against a baseline of a consensus-based token transfer system. This baseline uses BFT-Smart, a state-of-the-art state machine replication (SMR) protocol [BSA14]. Both Astro and BFT-Smart provide Byzantine fault-tolerance (BFT), assuming that less than one third of system nodes are Byzantine.

In our performance evaluation, we conduct experiments on a wide-area network using systems of up to 100 nodes. Depending on system size, we achieve a throughput improvement ranging from $1.5x$ to $6x$, while reaching up to $2x$ lower latency. Compared to consensus-based implementations, Astro is simpler and it does not depend on synchrony assumptions for progress.

We organize the rest of this chapter as follows. We start by providing an overview of the Exa abstraction and how we can avoid total order (§7.2). We then present the algorithms behind Astro (§7.3), as well as its implementations and evaluation (§7.4). We also discuss the contributions of this chapter in the context of related work (§7.5), and then conclude (§7.6).

7.2 Overview

The goal of the Exa abstraction is to provide support for efficient implementations of token-based transfer systems. We now describe this abstraction in more detail and discuss how we can implement it without employing total order.

7.2.1 The Exa Abstraction

An Exa object has a simple interface, which we describe in Listing 7.1, assuming a concrete object i. Each Exa object has an *owner* and an *identity*. For simplicity sake, we denote by i both an account (a concrete Exa object) as well as the identity of this object. The *state* attribute of the object represents the current amount of tokens, i.e., the balance.

```
1 // Attribute definitions:
2 owner        // Owner identity
3 identity     // Object (i.e., account) identity 'i', immutable
4 state        // Retains the current amount (balance) of tokens

6 // Operation definitions:
7 i.transfer(j, x) : returns {true, false}
8 i.getBalance()   : returns state
9 i.getOwner()     : returns owner
```

Listing 7.1 – The interface of an Exa object, instance i.

An Exa object i exposes an operation allowing the owner of this account to *transfer* an amount x of tokens to another account j (line 7). By convention, we say that this transfer is *outgoing* from account i and *incoming* to account j. A transfer operation succeeds, i.e., returns true, if and only if the balance on the outgoing account i has at least x tokens; in this case, the balance of account i diminishes by x and the balance of account j gains x tokens. Otherwise (if the balance of i is insufficient), the transfer operation fails, i.e., returns false, and the account balances remain unchanged.

Each Exa account also exposes operations for checking the current balance of the account (line 8) and reading the identity of the owner (line 9). Any user in the system may invoke these two operations, unlike the transfer operation.

7.2.2 Implementing Exa Objects Without Total Order

The central idea behind the Exa abstraction is that solely the owner of an account i may invoke the transfer operation (i.e., spend tokens) for this account. Given this access control policy on the transfer operation, the problem of ordering modifications to this replicated object (which typically boils down to solving consensus) reduces to the simple ordering of messages broadcast from a designated sender—the owner of the object.

To understand why total order is not necessary with Exa, consider a set of users who transfer tokens between their accounts. Assume the full replication model: Every user maintains a copy of the state of every account in the system. Notice that most transfer operations *commute*, i.e., different users can apply them in arbitrary order, resulting in the same final state. For instance, a transfer T_1 from the account of Alice to that of Bob commutes with a transfer T_2 from Carol's account to Drake's account. This is because these two transfers involve different accounts. In the absence of other transfers, each user can apply T_1 and T_2 on their state in different orders, without affecting correctness.

Consider now a more interesting case where two transfers involve the same account. For example, let us throw into the mix a transfer T_0 from Alice to Carol. Assume that Alice issues T_0 before she issues T_1. Note that T_0 and T_1 do not commute, because they involve the same account—that of Alice—and it is possible that she cannot fulfill both T_0 and T_1 (due to insufficient balance). We say that T_1 depends on T_0, and so T_0 should be applied before T_1.

Furthermore, suppose that Carol does not have enough tokens in her account to fulfill T_2 before she receives transfer T_0 from Alice. In this case, transfer T_2 depends on T_0. Thus, all users should apply T_0 before applying T_2, while transfers T_1 and T_2 still commute. The partial ordering among these three transfers is in fact given by a causality relationship. In Figure 7.1 we show the scenario with these three transfers and the dependencies between them.

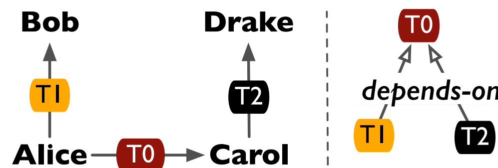


Figure 7.1 – Dependencies among transfer operations in a token-based system. On the left-hand side there are three transfers among four accounts, and on the right-hand side we depict the dependencies among these transfers.

Intuitively, the ordering constraint we seek to enforce among transfers is that every outgoing transfer for an account causally depends on all preceding transfers involving that—and only

that—account. To enforce this constraint in Astro, each user broadcasts their transfer operations using a primitive that ensures FIFO ordering across their messages. In other words, since each account has one owner, then for every account, all users in the system observe all transfers outgoing from that account in the same order. Additionally, users also specify any dependencies (i.e., incoming transfers), piggybacked with every transfer they broadcast. In the next section we discuss this broadcast primitive and how it fits into Astro, as well as the issue of tracking dependencies.

7.3 Astro System

In this section we focus on the Astro system. Before diving into details, we first discuss some preliminary notions (§7.3.1). Then we present Astro in full detail (§7.3.2) and briefly cover the Byzantine FIFO broadcast building block (§7.3.3).

7.3.1 Preliminaries

Astro is essentially a replicated system running at N nodes (i.e., replicas) in an asynchronous network. We assume that less than $N/3$ of nodes are Byzantine. This last assumption is important for securing our broadcast algorithm against Byzantine faults (§7.3.3).

We assume that each node in Astro is the owner of one account. Thus, the state of the system consists of N Exa objects. Intuitively, this system resembles a peer-to-peer model. We employ this unification of nodes with accounts for pedagogical reasons, as it makes it easier to reason about our algorithms. It is simple, however, to extend this model and allow multiple accounts per node, for instance, by generating multiple logical identities per node, one for for every owned account. Alternatively, we can introduce a mapping between nodes and accounts; in fact, this is precisely what we will do in our evaluation (§7.4.2.)

To simplify matters, we use the familiar notation i to identity not only a certain account in the system, but also the owner of that account, i.e., one of the nodes in the system. Furthermore, in the context of an operation i.transfer(j, x), we denote account i as the *spender*, while account j is the *beneficiary* of the transfer.

As explained, the essential property of an Exa object is that it has a unique owner, and we disallow non-owners from initiating transfers. We assume a permissioned setting (§1.1) where all nodes know each other, and we identify each node by its public key. Upon initiating a transfer operation, each node signs the details of this operation before broadcasting to others. Based on this authentication method, it is straightforward to restrict access by verifying signatures prior to applying any transfer on an Exa object.

7.3.2 Astro Algorithms

At a high level, Astro works as follows. To perform a transfer, a node i broadcasts a message with the transfer arguments, including the outgoing—or spender—account (in this case account i), the beneficiary account, and the amount of tokens. These arguments match with the transfer operation in the Exa interface presented earlier (Listing 7.1, line 7).

Recall that a correct node should not apply a transfer T before applying all the dependencies for T. In particular, transfers outgoing from the same account i do not commute, and thus all correct nodes must apply these transfers in the same order. This FIFO property is simple to enforce in practice by relying on sequence numbers. Concretely, our Astro algorithm builds atop a Byzantine FIFO broadcast primitive, which we cover later (§7.3.3).

By itself, Byzantine FIFO broadcast is not sufficient to obtain the causal ordering guarantees we seek. Recall there is an additional scenario in which two transfers do not commute (§7.2.2). Suppose that account i has initially zero balance and receives some tokens in a transfer T_1. Thereafter, account i immediately sends tokens to account j in a transfer T_2. If any node in the system tries to apply T_2 before T_1, it would consider the former transfer invalid due to insufficient funds. Thus, transfer T_2 depends on T_1 and all nodes in the system should apply T_1 before T_2. We enforce this ordering by attaching a dependency set to every transfer operation. Before applying any transfer, we require every node to apply the history attached to that transfer. For performance reasons, in practice the history attached to a transfer need not contain other transfers, but merely their identifiers, in a similar vein as vector clocks represent causal history in a classic causal order broadcast [Fid88, Mat88].

Before proceeding to explain Astro in more detail, we first make a refinement on the system model, to introduce the abstraction of committees.

Committees. Throughout this chapter so far we assumed the full replication model. In this model, each node i broadcasts the details of all their transfer operations to the whole system. This allows every node to replicate all accounts. We describe a more flexible design, which we call a committee-based model, inspired from partial replication schemes.

In the committee-based model, we associate each account i to a certain set of system nodes. This set is the committee of i. Intuitively, the committee of an account witnesses all transfers outgoing from that account and guarantees their correctness. The restriction of less than $N/3$ of Byzantine nodes we mentioned earlier applies to every committee in the system. (This is because each committee runs an instance of Byzantine FIFO broadcast.)

This design based on committees is a simple generalization of the full replication model (where the committee of every account is the whole system). Note that the committees of two accounts may completely overlap or may be completely disjoint. We introduce this abstraction of committees to enable a more modular design, open to future extensions. We further

```
10 sn[1..N]     // Counts how many outgoing transfers have been applied for each account
11 hist[1..N]   // List of applied transfers for every account
12 deps         // Set of delivered transfers that involve the local account i
13 buffer       // Incomplete dependencies
14 com[1..N]    // The committee of each account
15 pending      // Set of transfers delivered (but not applied)
```

Listing 7.2 – The state at each node *i* in Astro.

argue for the value of this separation in a later discussion (§7.5).

Node state. In Listing 7.2 we describe the local state from the perspective of a particular node *i*. Each node keeps, for each Exa object *j* in the system, an integer sequence number *sn[j]* reflecting the number of transfers outgoing from account *j* which the local node *i* has applied. Each node in Astro also maintains a list *hist[j]* of transfers which involve account *j* and local node *i* has applied. We say that a transfer operation involves an account *j* if that transfer is either outgoing or incoming on account *j*.

Each node *i* maintains a local variable *deps*, representing dependencies for their own Exa object. More precisely, this variable is a set of transfers having account *i* as beneficiary, which node *i* has applied since the last successful (outgoing) transfer. The *buffer* variable temporarily stores incomplete dependencies before they are ready to be used.

Finally, each node also maintains the committee of each account in the system, as well as a variable *pending*, which is a set containing delivered transfers that require validation, i.e., have not been validated nor applied yet on the state.

Initiating a transfer of tokens. We sketch the algorithm for this operation in Listing 7.3 below. To transfer tokens, node *i* first checks the balance of its own account (line 17). If there is not enough funding, i.e., the balance is insufficient, this operation returns false (line 18). Otherwise, node *i* broadcasts to the committee of its account a *transfer* message *m* (line 19), encoding the details of this operation via the broadcast primitive (line 20).

```
16 func i.transfer(j, x)
17   if (i.getBalance() < x)
18     return false
19   m := ⟨TRANSFER, i, j, x, sn[i] + 1, deps⟩
20   broadcast(com[i], m) // Broadcast to my committee
21   deps := {} // Reset my dependencies
22   return true
```

Listing 7.3 – Algorithm for the transfer operation in Astro. Code for account *i*.

```
23 callback deliverBroadcast(j, m)
24   if (i ∉ com[j])
25     return // Drop message if we are not in committee of 'j'
26   pending := pending ∪ {(j, m)}
```

Listing 7.4 – Byzantine FIFO broadcast callback: Node i delivers a message m broadcast by a node j.

The message m which node i broadcasts to its committee includes the three basic arguments of the transfer operation, plus the sequence number for the spender account *sn[i]*, as well as the accumulated dependencies (line 19). After broadcasting m, node i empties its set of dependencies, and returns true (line 22).

Delivering and applying a transfer. The nodes in the committee of an account deliver broadcast messages via a callback from the broadcast primitive. Listing 7.4 presents this callback. Assume node i delivers a broadcast message m, sent by a node j. Upon delivery, node i first checks that it belongs to the committee of the sending node j (line 24). If this check passes, then node i saves message m in the *pending* set (line 26), for validation. We explain the validation procedure later.

Applying a valid transfer. The algorithm in Listing 7.5 describes how a node i applies a transfer operation after that operation passes validation. Assume account j is the spender in this transfer (line 28). First, node i adds transfer t as well as its dependencies to the history of Exa object j (line 30). Then node i also updates the sequence number for j (line 31).

```
27 upon (j, m) ∈ pending and valid(j,m)
28   let m be ⟨TRANSFER, j, k, x, n, D⟩
29   t := (j, k, x, n)
30   hist[j] := hist[j] ∪ t ∪ D
31   sn[j] := n
32   // Finished applying the transfer
33   proof := ⟨PROOF, t, Sign(t)⟩ // Prepare a proof message
34   unicast(k, proof) // Inform the beneficiary that it received tokens
```

Listing 7.5 – Node i (belonging to committee of account j) applies a transfer operation where account j is the spender and account k is beneficiary.

In Listing 7.5 observe that the beneficiary of transfer t is some account k (line 29). Node k which owns this Exa object does not necessarily belong to the committee of the spender account j. Hence node k does not necessarily deliver and apply transfer t on its local state. The mechanism for informing k that it is the beneficiary of some tokens (in a transfer from account j) consists of a message which every committee member of spender j unicasts to k.

We call such a message a *proof* for transfer t. A proof consists of the transfer details as well as a signature on the transfer (lines 33–34). We describe next how the beneficiary handles the delivery of proofs and afterwards we cover the *valid* procedure for validating transfer messages.

Handling transfers at the beneficiary. Whenever some account k is the beneficiary of a certain transfer operation t, the node owning account k expects proofs for this transfer. Every correct node that applies t sends such a proof to node k. Upon gathering sufficiently many proofs for transfer t, this transfer becomes a *dependency* for account k. The set of proofs are called a *certificate* for that dependency. Intuitively, a dependency represents some unused tokens, while the corresponding certificate proves the validity of that dependency.

The algorithm in Listing 7.6 below shows the callback which a node i uses to deliver a proof message for a transfer t. Notice that the sender of this proof is node j, and assume account k is the beneficiary of this transfer, while account l is the spender (line 37).

```
35 callback deliverUnicast(j, proof)
36   let proof be ⟨PROOF, t, sig⟩
37   let t be (l, k, x, n)
38   if (k = i)
39       and (j ∈ com[l])
40       and checkSig(j, t, sig)
41     buffer[t] := buffer[t] ∪ {j, sig} // Accept correct proof
42   if |buffer[t]| = faultThreshold + 1 // F+1
43     // Gathered sufficient proofs to form a certificate
44     deps := deps ∪ {t, buffer[t]}  // Transfer becomes a full−fledged dependency
45     buffer[t] := {}
```

Listing 7.6 – Handling the delivery of a proof message for a transfer t, sent by node j.

When node i delivers a proof, three verifications are necessary to ensure that the proof is correct, as described below.

1. The beneficiary account reported in the proof message (labeled k on line 37) should match the local account i (line 38);
2. The sender j of the proof should be in the committee of the spender l (line 39); and
3. The signature in the proof should correspond to sender j (line 40).

After ensuring that a proof message for transfer t is correct, node i adds this proof to the *buffer* set. Note that transfer t is not yet a dependency ready to be used. This *buffer* set stores proofs for different transfers temporarily. To become a dependency, a transfer must have

$F+1$ matching correct proofs (this is the *faultThreshold* variable on line 42). Once sufficient proofs are gathered from different committee members of spender account l, these proofs form a certificate guaranteeing the correctness of that transfer. Then transfer t becomes a dependency for the local account i (line 44), ready to be used in later transfers.

Validating a transfer and its dependencies. As explained earlier, before applying a transfer, each node first validates it via the *valid* function (line 27). We describe the validity conditions that a transfer has to fulfill in Listing 7.7.

```
46 func valid(j, m)
47   let m be ⟨TRANSFER, l, k, x, n, D⟩
48   return (l = j)
49          and (n = sn[j] + 1)
50          and (checkDeps(j, D))
51          and (computeBalance(j, hist[j] ∪ D) ≥ x)
```

Listing 7.7 – Validity checks for a transfer message m sent by node j.

To be considered valid, a transfer message m must satisfy four conditions (lines 48–51). The first condition is that node j (the sender of this transfer message) must be the owner of the outgoing account, i.e., the spender in this transfer (line 48). Second, any preceding transfers that node j issued from their account must have been validated (line 49). Third, every declared dependency for this transfer (and its certificate) must be valid (line 50); we handle this step in a separate procedure *checkDeps*, discussed below. Fourth, the balance of Exa object j must not drop below zero (line 51).

```
52 func checkDeps(j, D)
53   foreach (t, cert) ∈ D
54     let t be (l, k, x, n)
55     if (|cert| ≤ faultThreshold)
56         or (k != j)
57       return false
58     foreach (s, sig) ∈ cert  // Check each proof in the certificate
59       if (s ∉ com[l])
60           or (! checkSig(s, t, sig))
61         return false
62   return true
```

Listing 7.8 – Verification of dependencies and their certificate.

We use the *checkDeps* procedure in Listing 7.8 to validate a set D of dependencies. For every dependency, this procedure does three checks, as follows.

1. The certificate for that dependency must be sufficiently large, i.e., its size must exceed a certain threshold, called *faultThreshold* on line 55.

2. The sender of these dependencies (node *j*) must be the beneficiary of each dependency, as checked on line 56.

3. For every dependency, the proofs in the certificate of that dependency must originate from a committee member (line 59) and must be correctly signed (line 60). Note that this check assumes that each signature originates from a distinct committee member, i.e., the set *certs* disallows multiple proofs from the same sender.

Reading the balance of an account. To obtain the balance of an Exa object, as shown in Listing 7.9, we employ a procedure *computeBalance*. We also use this same procedure in the *valid* function (line 51). We omit the internals of this procedure, which computes the current balance of account *i* by traversing the history for this Exa object and its current dependencies, summing up all the incoming transfers and subtracting the outgoing transfers for *i*.

```
63 func i.getBalance()
64   return computeBalance(i, hist[i] ∪ deps)
```

Listing 7.9 – Computing the balance of account *i*.

For a discussion on the correctness of Astro, we refer the interested reader to Appendix B.1.

7.3.3 Overview of Byzantine FIFO Broadcast

In a nutshell, Byzantine broadcast guarantees that messages broadcast by a correct node are eventually delivered by every correct node. The FIFO property, in addition, says that if a correct node broadcasts some message *m* and then broadcasts another message *m'*, then no correct node delivers *m'* before delivering *m*.

We now sketch a simple algorithm for implementing Byzantine FIFO broadcast, inspired from prior work on a similar primitive called double-echo broadcast [BT85, CGR11]. In this description we ignore the concept of committees, and adopt a model where messages are broadcast to all *N* nodes in the system. We assume the number *F* of Byzantine nodes is less than one-third of *N*. All correct nodes in this algorithm communicate via authenticated links. At a high-level, this algorithm comprises three phases, as follows:

1. **Send**—To broadcast a message *m*, a correct node *i* attaches a sequence number *s* to this message, forming a tuple *(m,s)*. Then node *i* sends this tuple to all nodes.

2. **Echo**—Upon receiving the tuple *(m,s)* from *i*, every node *j* sends an echo message of this tuple to all nodes in the system.

3. **Ready**—When a node j gathers a Byzantine quorum of echo messages for the tuple *(m,s)*, then node j sends a ready message for this tuple to all nodes. Each node delivers message m after obtaining ready messages from $2F+1$ distinct nodes, and after having delivered all messages from node i that have sequence numbers smaller than s.

The full algorithm for implementing Byzantine FIFO broadcast is out of the scope of this dissertation, hence we defer it to the appendix (Appendix B.2).

7.4 Experimental Evaluation

In this section we experimentally evaluate Astro, our token transfer system implementing the Exa abstraction. We first describe the two systems we use in this evaluation, namely (1) our implementation of Astro, and (2) a transfer system based on consensus (§7.4.1). We then present our experimental methodology (§7.4.2), and present our results (§7.4.3).

7.4.1 Transfer Systems Under Evaluation

We implement a prototype of Astro in the Go programming language using roughly $1.6K$ LOC (lines of code). Our implementation builds on a Byzantine FIFO broadcast primitive inspired from the Asynchronous Byzantine Agreement (ABA) of Bracha and Toueg [BT85]. Note that it is simple to build Byzantine FIFO broadcast starting from an ABA building block by relying on sequence numbers [Rei94], which are needed to ensure the FIFO property. We reuse the sequence numbers of this broadcast layer to tag each transfer from a given node (the *sn* variable in broadcast messages, line 19 of Listing 7.3), which helps mildly decrease the total size of messages in our implementation.

To obtain authenticated links at the Byzantine FIFO broadcast layer, we rely on Message Authentication Codes (MACs) between every pair of nodes, implemented using SHA256. We employ several standard optimizations in this implementation, such as batching and pipelining [SS12]. Such optimizations exist as well in our baseline system, which we describe next.

The goal is to compare our implementation against a baseline transfer system that builds on consensus. We build this baseline on top of BFT-Smart [BSA14]. This is a state-of-the-art implementation of SMR with Byzantine fault-tolerance guarantees. BFT-Smart is currently being used with success in the digital trust ecosystem, e.g., in the Hyperledger Fabric project [SBV18]. For both our baseline and Astro, we assume the optimal threshold of $N = 3F+1$ replicas, where F is the upper bound on the number of replicas that can be Byzantine. Throughout our evaluation, we will adopt the SMR terminology and refer to protocol participants as *replicas*. We seek to understand how a committee running the Astro algorithm compares with a committee running BFT-Smart, i.e., the rest of this evaluation will focus on systems consisting of a single committee of N replicas.

It is important to note that Astro (implemented on top of broadcast) is significantly simpler than consensus-based solutions. In general, there is a staggering difference in complexity between consensus—in particular BFT—and broadcast algorithms. Contrast our $1.6K$ LOC implementation, for instance, with `libpaxos` [Lib18], which is a simple implementation of Paxos for the crash-only system model. This Paxos codebase stretches over more than $6K$ LOC in C (without counting the event-based library on which it relies). At the time of its original publication, the BFT-Smart implementation counted around $13.5K$ LOC in Java [BSA14, §III].

Astro is an asynchronous algorithm, meaning that our implementation does not have to rely inherently on timeouts to ensure liveness. This is in contrast to deterministic consensus-based solutions. Such solutions have to resort to fine-tuned timeout parameters which can affect their performance and availability, as our experiments show later (§7.4.4). This is a well-known aspect that has been reported before [BSA14, CWA+09, DRZ18, MXC+16].

7.4.2 Methodology

We use Amazon EC2 as our experimental platform. Throughout the experiments, we use commodity-level virtual machines of type `t2.medium` [Ec218]. We deploy these systems so that each replica executes on a separate virtual machine. This helps avoid creating any unnecessary noise in our results, which could arise due to performance interference.

We consider two deployment scenarios:

1. The first scenario is *continental*, comprising four regions in Europe, namely Frankfurt, Ireland, London, and Paris.
2. The second scenario we use is *global*. In this scenario, we use ten regions around the globe, namely Frankfurt, Ireland, London, N. Virginia, Ohio, Oregon, Singapore, Canada, Mumbai, and Sydney.

In both scenarios, we deploy the replicas of each of the two systems randomly across the corresponding regions. The goal of having these two separate scenarios is to evaluate whether different wide-area deployments affect the behavior of the systems we are evaluating. The first scenario reflects an inter-banking transfer system, for instance, where participants are localized in a single geographic region—namely in Europe—while the latter scenario reflects a deployment where participants are spread around the whole world.

As we have described our system so far, in Astro there is no separate client role, and the workload of the system originates directly from every participating node, i.e., replica (§7.3). In contrast, most SMR implementations, including BFT-Smart, assume a client-server architecture. To be fair towards BFT-Smart, we introduce the role of clients in Astro; these are separate nodes, each connecting to one of the replicas in the system. Introducing clients allows us to

obtain an apples-to-apples comparison of latencies and throughput between Astro and the consensus-based implementation.

In both the global and continental scenarios we deploy up to 15 machines for client processes, which generate the workload. Each request from a client represents one transfer operation. A request contains three fields: spender identity, beneficiary identity, and the amount. By default, all three fields in a transfer operation have random values. The only constraint is that the spender in a given transfer operation is always an account associated with the replica where that operation is being invoked. In other words, the replica which handles a transfer is the owner of the spender (outgoing) account in that transfer.

For simplicity, we place all client machines in Ireland. Spreading clients around the world does not influence the results of our evaluation. Each client machine hosts a varying number of client processes. The number of processes varies greatly, depending on each system and the system size. For instance, to saturate throughput in BFT-Smart at system size $N = 4$, we use around 800 client threads; for the same system at $N = 100$, 30 threads are sufficient to reach saturation. For Astro, we require more client threads to reach saturation, since this system is capable of higher throughput.

For throughput, we report on how many transfers each system executes per second. We are interested in the peak throughput each system can sustain as a function of system size. All of the experiments we discuss consist of a runtime of 60 seconds, and we present the average result across 3 runs, eliding a warm-up and a cool-down of 10 seconds each. We also plot the standard deviation, but often this has negligible values and is not clearly visible in the plots.

In BFT-Smart, each client keeps connections to all replicas. This is a design choice of this system [BSA14]. For this reason, all BFT-Smart clients experience similar latencies when executing their workload. The latency we report is the one observed by a random client.

Unlike BFT-Smart (where there is no ownership relation between accounts and replicas), in Astro each account is uniquely associated to a replica. This is a property of the Exa abstraction, so it is the owner replica that performs transfers on their respective accounts. It is only for comparison with BFT-Smart that we add the client layer, creating the situation where replicas execute transfers "on behalf of the clients". To stress the system and make all replicas execute transfers (which is the most realistic scenario), each client connects and submits the whole workload to some random replica. The performance of Astro remains the same, however, even if all clients connect to a single replica; we discuss this in further detail later.

7.4.3 Performance Evaluation Results

We first discuss the results for throughput and then show how the two systems compare in terms of latency. We also show that load-balancing, i.e., the assignment of clients to replicas, does not impact the performance of Astro.

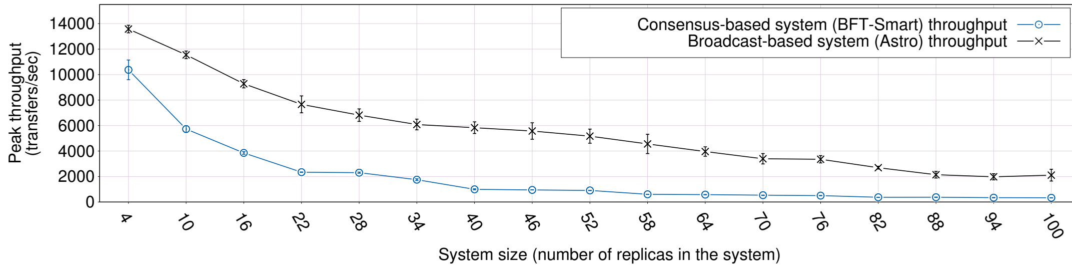


Figure 7.2 – *Regional scenario* (using four regions across Europe, namely Frankfurt, Ireland, London, and Paris). We report on peak throughput as we increase the number of replicas in two transfer system implementations, one based on broadcast, and another based on consensus (i.e., SMR).

Throughput

In Figure 7.2 we present the results for the continental scenario. We depict how throughput evolves as a function of system size, i.e., for each system size, we plot the peak throughput. We start from the smallest system size, $N = 4$, and subsequently add more replicas. We increase the size of each system in increments of 6, until we reach 100 replicas.

At a small system size, both systems exhibit the highest throughput. The consensus-based implementation using BFT-Smart sustains over $10K$ transfers/sec, while Astro reaches almost $13.5K$ transfers/sec. As we increase the system size, throughput degrades considerably. The consensus-based system saturates at 334 transfers/sec at $N = 100$. In contrast, Astro can sustain $6X$ higher throughput, being able to apply $2K$ transfers/sec at a system size of 100 replicas. Notice that these results are in line with our earlier findings on the performance decay of SMR from Chapter 3, Figure 3.1. This is despite different testbeds, though they have similar characteristics (EC2 in this case, versus SoftLayer for earlier experiments §3.3.1).

The results for the global scenario are depicted in Figure 7.3. Note that in this scenario the smallest system size we consider is $N = 10$. This is because this scenario comprises 10 separate regions in the world and we want to cover each region with at least one replica. We proceed in increments of 10, so that at each subsequent iteration we add one additional replica to each region until we reach 100 total replicas.

Similarly to the continental scenario, throughput degrades considerably with growing system size. The degradation in BFT-Smart is less sharp than in Astro, but this is partly because the former system starts from a significantly lower throughput at a small scale.

In this scenario, BFT-Smart exhibits a throughput of $1.5K$ transfers/sec when deployed on ten replicas, and it can only reach 242 transfers/sec at the largest system size. Astro sustains $9.4K$ transfers/sec when deployed on ten replicas. When running on $N = 100$ replicas, Astro reaches 357 transfers/sec, which is roughly 50% increase over the consensus-based solution. In absolute numbers, throughput in the global scenario is significantly smaller than in the

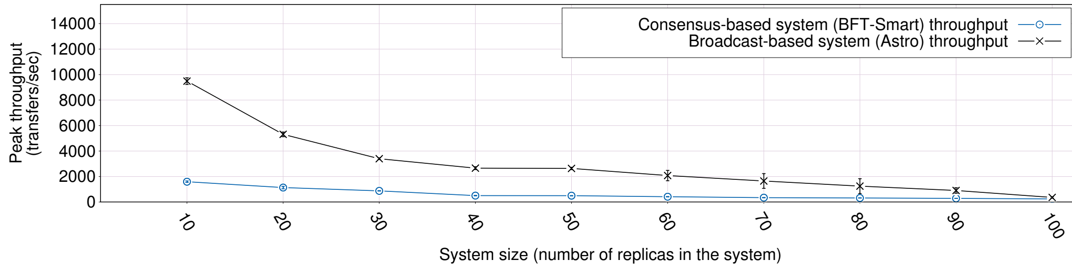


Figure 7.3 – *Global scenario* (using ten regions across the globe: Frankfurt, Ireland, London, N. Virginia, Ohio, Oregon, Singapore, Canada, Mumbai, and Sydney). We report on peak throughput for two transfer system implementations, namely one based on Byzantine FIFO broadcast, and another based on consensus (i.e., SMR).

continental case. This is mainly due to the difference in the network bandwidth between the two configurations. Concretely, the bandwidth available across regions in Europe is typically $3x - 5x$ larger than the bandwidth when crossing from regions in Europe to Southeast Asia.

In terms of throughput, our Astro prototype outperforms the consensus-based solution at every system size up to 100 replicas. We did not experiment beyond this system size, mainly due to resource (economical) constraints. We expect, however, that eventually, at a larger system size the two systems will exhibit similar throughput. Even if Astro is broadcast-based, this system still relies on quorum-gathering techniques to achieve consistent replication, and hence the throughput decay in this system follows the same decay as consensus-based solutions.

A practical deployment of Astro could consist of a certain number of banks, for instance, seeking to build a decentralized inter-bank transfer (or, more generally, asset exchange) system. If we consider that each participant is an individual bank, then a solution based on our consensusless algorithm can handle, for instance, almost $3K$ transfers/sec when there are 50 participating banks in the global setting and can handle almost $6K$ transfers/sec for the same number of participants in the continental setting. In a nutshell, for systems of moderate size—up to 100 replicas—Astro is simpler and significantly outperforms consensus-based solutions for decentralized transfers.

Latency

In Figures 7.4 and 7.5 we depict how latency evolves with respect to throughput in both systems, for two particular system sizes, namely $N = 10$ and $N = 100$, respectively. Note the different scale of the x-axis across these two figures. These results are for the global scenario (this is where the two systems are at their worst). We plot both average latency and the $95th$ percentile latency.

For the small system size (Figure 7.4), when the workload is light, the consensus-based sys-

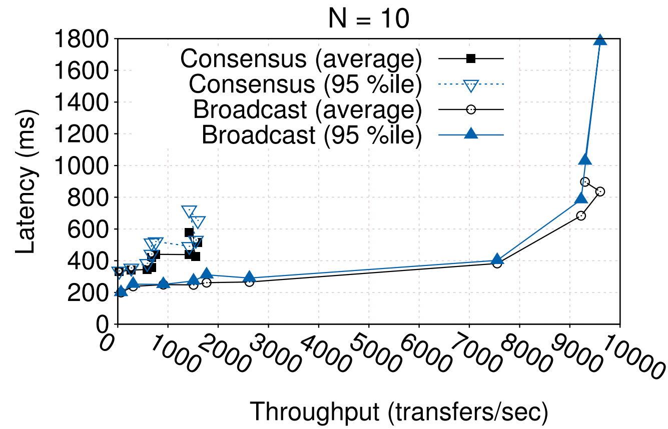


Figure 7.4 – Throughput-latency graph for $N = 10$, comparing Astro against a consensus-based solution. The deployment environment is global.

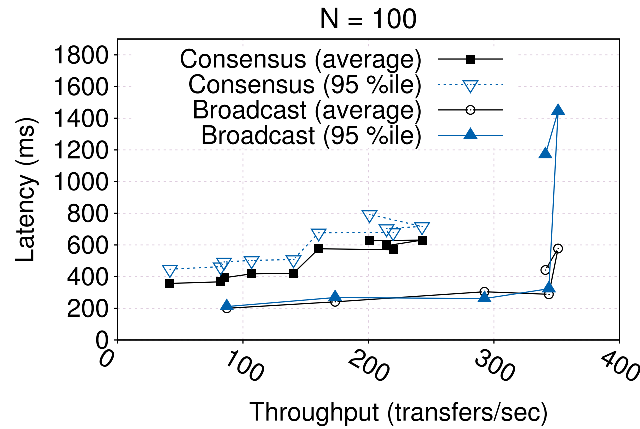


Figure 7.5 – Throughput-latency graph for $N = 100$, comparing Astro against a consensus-based solution. The deployment environment is global.

tem has an average latency of roughly $330ms$, while the broadcast-based system exhibits on average $200ms$ latency. The $95th$ percentile latencies at low load are similar to the average latencies for both systems.

We observe that the latency remains almost unchanged when we increase the system size from $N = 10$ to $N = 100$ (Figure 7.5). Concretely, when the systems are not under high load, the average latency is $350ms$ for the consensus-based system and $200ms$ for Astro. This is unsurprising, however. Recall that the deployment characteristics for these sets of results are the same—i.e., we use the global deployment—for both the small ($N = 10$ in Figure 7.4) and the large system ($N = 100$ in Figure 7.5). For both system sizes, the replicas are geographically distributed in 10 regions of the world. The only difference between the two experiments is the number of replicas at each region. The latencies do not change considerably because obtaining one response from a particular region takes roughly as much time as obtaining

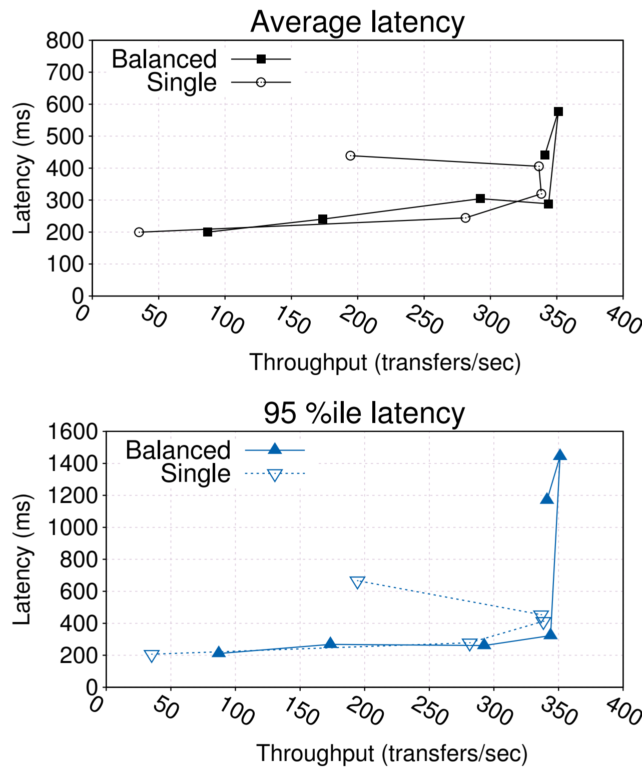


Figure 7.6 – Throughput-latency graph for balanced versus single setups in Astro. We use $N = 100$ replicas in the global environment.

several responses (in parallel) from that region.

Workload balancing among replicas

In this experiment we examine how the balancing of clients across replicas affects the performance in Astro. To do so, we deploy our token transfer system on 100 replicas in the global environment. We contrast the difference in performance between the following two setups:

1. *Balanced:* where we assign clients to random replicas across the system. This is the strategy that we used throughout all experiments so far.

2. *Single:* where we assign all clients to one single replica.

We study how latency—average and $95th$ percentile—evolves as a function of throughput. The results are depicted in Figure 7.6. To enhance visibility, we plot the average latency separately (in the top graph) from the $95th$ percentile (which is in the bottom graph). Note that the y-axis is different across the two pictures, to account for higher $95th$ percentile latencies.

Unsurprisingly, in both the balanced and single setups, the throughput saturates at roughly the same point, around 350 transfers/sec. These numbers are consistent with previous observations we made (Figures 7.3 and 7.5). Latency is also consistent with previous findings, and it fluctuates between 200 and 300ms.

We can conclude that inserting transfers in the system (in which only one replica is involved) does not require a significant overhead compared to the overhead of broadcasting these transfers (in which all replicas are involved). Another insightful way of interpreting this result is that no single replica in Astro is the bottleneck.

7.4.4 Availability Evaluation Results

We now investigate how these systems react to two problems that can arise in practice, namely *failure* (e.g., crash) and *asynchrony* (network delays) at a replica. We consider the impact of these issues when they affect a random replica in each system, as well as the case when the leader is affected in the Consensus-based system. The role of a leader does not exists in Astro.

We study the evolution of throughput within a window of execution of 40 seconds, ignoring a warm-up period of 20 seconds. For all these experiments, we introduce asynchrony or failure after 30 seconds elapse. To induce asynchrony, we use the traffic control utility `tc` with the network emulator queuing discipline. We always use a delay of 100ms. For instance, to introduce such a delay on all packets outgoing from interface `eth0` at a replica, we use the following command: `tc qdisc change dev eth0 root netem delay 100ms`.

We use 10 clients, each running a single thread. The goal is to evaluate these systems below saturation point. If we introduce failures at saturation, this can lead BFT-Smart to halt or enter a livelock where the system is unable to do view-changes (i.e. re-elect leader). Moreover, at saturation point the broadcast-based system can sustain the same throughput independently of how many replicas accept client operations, as we showed in previous experiments (Figure 7.6). In other words, stopping a replica at saturation point in Astro should not impact

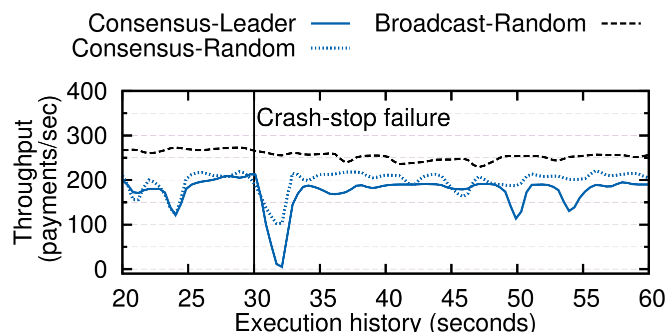


Figure 7.7 – Throughput evolution when we introduce a crash-stop failure of a single replica in the Consensus-based system (either the leader or a random replica) and the Astro broadcast-based system (a random replica).

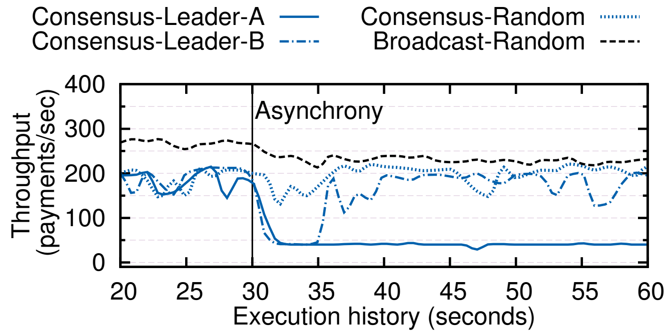


Figure 7.8 – Throughput evolution in the consensus- and broadcast-based systems when we introduce asynchrony (100*ms* delay for each outgoing packet) at a replica, either the leader or a random one.

throughput, giving an advantage to our system over the Consensus-based solution. We first report results for $N = 49$ in the *regional* configuration. We run these experiments with larger and smaller systems, but similar observations emerge as the ones we describe below. For completeness, we also discuss a set of interesting results with a larger system size of $N = 100$.

In Figure 7.7 we show the how throughput evolves when we introduce a crash-stop failure at a replica ($N = 49$). For consensus, this failure has a severe impact on throughput if the leader is affected (the *Consensus-Leader* timeline), because the view-change protocol has to execute. The throughput drops to 0 while this protocol runs, typically a few seconds. For larger system sizes, this protocol can take longer to execute, as we will show later. When a random replica fails in the consensus-based system (*Consensus-Random*), there is a brief decrease in throughput when all clients and replicas get disconnected from the affected replica, but thereafter performance recovers.

When we stop a random replica in the broadcast-based system (*Broadcast-Random* in Figure 7.7), throughput drops from 270 ops/sec to 250 ops/sec. This accounts for the failed replica which was handling roughly 20 operations per second from one of the clients. This decrease is barely visible in the plots.

Figure 7.8 shows how asynchrony impacts the performance in the two systems ($N = 49$). We depict two separate executions for the case of consensus when the leader is affected, because there are two possible outcomes. First, it may happen that throughput decreases and remains that way; this is the *Consensus-Leader-A* timeline. Second, the system can go through a view-change (*Consensus-Leader-B*) because the leader is too slow or its buffers can overflow and packets get dropped (inflating the replica-to-replica delay). Clearly, initiating a view-change is preferable in this case, because the throughput penalty is smaller. There is a well-known tradeoff, however, in choosing the view-change timeout [CWA+09, MXC+16]: initiating view-change too aggressively can lead to frequent leader changes even in good conditions, which can erode performance on the long-run.

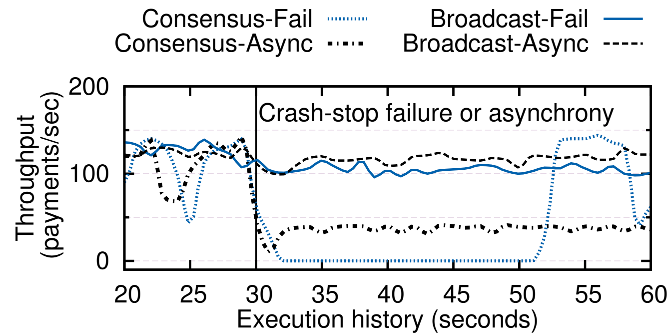


Figure 7.9 – Throughput evolution for $N = 100$ when a crash-stop failure or asynchrony affects the consensus-based system or the broadcast-based system.

When a random replica is affected with asynchrony in the consensus-based system (*Consensus-Random* line in Figure 7.8), performance drops briefly because there is a quorum switch, i.e., the affected replica is replaced by a different one in the quorum [AGMS18]. For the broadcast-based system (the *Broadcast-Random* timeline), asynchrony affects performance in the same manner in which a failure does. Concretely, the affected replica no longer sustains the same amount of client operations, so the overall throughput reduces correspondingly.

We also show results for the case of a larger system size ($N = 100$) in Figure 7.9. There are four timelines in this execution, as follows. For the consensus-based solution, we show what happens when there is either a crash-stop failure or asynchrony at the leader. In the former case (*Consensus-Fail*), the view-change protocol kicks in and lasts for roughly 20 seconds, while throughput stays at zero; this is similar to the *Consensus-Leader* execution in Figure 7.7. In the latter case (*Consensus-Async*), performance degrades and stays that way for as long as the affected replica remains the leader; this is similar to the *Consensus-Leader-A* in Figure 7.8. For the broadcast-based solution we consider the same two issues affecting a random replica. When either of these issues arises (*Broadcast-Fail* or *Broadcast-Async*) throughput is affected correspondingly with the number of operations that the failed replica is handling (and which is unable to continue). Being a peer-to-peer system, in Astro there is fate-sharing [Cla88] between each replica in the system and the account(s) hosted at that replica. For this reason, when a replica stops, all the associated accounts naturally stop executing operations as well.

We wrap-up this section with two general observations. First, Astro does not suffer from overall (i.e., global) throughput degradation than can happen in leader-based protocols such as most consensus algorithms. Second, our system does not need to rely on timeouts for liveness. Simply put, Astro progresses at the speed of the network. These two advantages are closely linked, and they both follow from the asynchronous nature of the broadcast protocol underlying Astro.

7.5 Discussion & Related Work

We omit several optimizations and limitations from our earlier description of Astro (§7.3). We discuss these now, while also covering related work.

7.5.1 State Sharding

In our evaluation of Astro all replicas participate (and replicate) every account. In other words, we employ the full replication model, where all Exa objects use the same committee. This reflects the same model our baseline BFT-Smart adopts. Given this setup, one important optimization that we employ in Astro is that we can skip the step of informing the beneficiary node for every transfer t, since this node is already part of the committee approving t. Furthermore, we can trim down the representation of dependencies in this setup. Recall that each dependency in our algorithm consists of transfer details plus a certificate proving the correctness of that transfer (§7.3.2). Since every node witnesses all transfers, a dependency can consist of a simple tuple $\{i, s\}$ pointing to a specific transfer with sequence number s of a specific Exa object i.

The full replication model entails storing the whole state and this is usually undesirable. A common alternative is to employ partial replication, often called sharding. Under partial replication, every user stores some segment of the state which is relevant for that user. The Astro design is amenable to partial replication, given the committee-based model we adopt. Concretely, each shard would replicate a subset of all Exa objects, effectively representing the committee of all these accounts. As an optimization, we imagine that we can satisfy certain locality constraints, e.g., an Exa object in Europe relying on a committee predominantly represented in Europe.

A problem that can appear in partial replication has to do with validating operations. In particular, it may happen that a user is unable to validate an operation if that operation depends on state which this user does not store locally. This is a well-known problem, both in systems which employ total order [ABK+15, FP18] as well as in those employing weaker ordering such as causality [BFG+12]. The purpose of a certificate which a node sends with every dependency is to solve this very issue in Astro. We remark on Corda, which proposes an interesting partial replication design for a blockchain (i.e., digital trust), allowing a user to resolve and validate dependencies by downloading any missing state directly from the user involved in each operation [Hea16, §4.2].

7.5.2 Account State Representation

Bitcoin and other cryptocurrencies do not directly deal with accounts and account balances. Instead, each outgoing transfer directly references one or more incoming transfers. It is important that each such incoming transfer is only referenced (i.e., spent) once. A node validates

new transfers by checking whether referenced transfers have not yet been spent. This is the unspent transaction output (UTXO) model [Nak08]. Keeping track of every transfer is feasible for low-throughput systems like Bitcoin, but doing so can be prohibitively expensive in terms of storage space for systems with higher throughput, because the state would grow very fast.

In Astro we keep track of all transfers and use a naive method of computing account balances only for simplicity of presentation (line 64 in Listing 7.9). In a practical deployment, the state of each account (the *hist* variable) can be represented not as a list of transfers, but as a simple pair of values: one for the account balance and one for the sequence number of the last applied outgoing transfer for that account. In this case, validating a dependency would require a modification, reducing to a check that the sequence number of a dependency is smaller or equal to the last applied sequence number for that Exa object.

7.5.3 Ordering Constraints

In the digital trust (i.e., blockchain) ecosystem, several efforts exist which avoid building a totally ordered chain of transfers. The idea is to replace the totally ordered linear structure of a blockchain with that of a directed acyclic graph (DAG) for structuring the transfers in the system. Notable systems in this spirit include Vegvisir [KJW+18], or Corda [Hea16]. Even if these systems use a DAG to replace the classic blockchain, their algorithms still employ consensus. As described, we can also use a DAG to characterize the relation between transfers in Astro, but our system relies on a broadcast primitive with BFT and FIFO ordering guarantees. Given the ownership feature of Exa objects, we maintain a per-account sequence of transfers. Each sequence is ordered individually—by its respective owner—and is loosely coupled with other sequences through dependencies established by causality.

We can draw a parallel between our Exa abstraction and conflict-free replicated data types (CRDTs) [SPBZ11]. Similar to a CRDT, in Astro we support concurrent updates on different Exa objects while preserving consistency. Each account has a unique owner, ruling out the possibility of conflicting operations when owners are correct. This ensures that the state at correct nodes converges. In the terminology of [SPBZ11], we provide strong eventual consistency.

The ordering we enforce in Astro is similar to one based on explicit causality [BFG+12]. In explicit causality, the dependency set for an operation are application-dependent. For instance, in a Twitter-like platform, when a user posts a new reply, the dependency set for this event can consist of only the original thread for that reply. In a similar vein, in Astro we specify that each transfer outgoing from an account only depends on previous transfers outgoing from and incoming to that—and only that—account.

7.5.4 Asynchronous Protocols

The insight that an asynchronous broadcast-style abstraction suffices for transfers appears in the literature as early as 2002, due to Pedone and Schiper [PS02]. Duan et. al. [DRZ18] intro-

duce efficient BFT protocols for storage and also build on this insight, as does recent work by Gupta [Gup16] on financial transfers which is the closest to the work in this chapter. We build on a recent theoretical result which shows formally that indeed consensus is unnecessary for transfers [GKM+18]; we exploit this result by proposing an abstraction and a practical system.

An important part of our implementation is the Byzantine FIFO broadcast primitive. This has its roots in the Asynchronous Byzantine Agreement (ABA) problem, defined by Bracha and Toueg [BT85]. There are multiple algorithms for implementing Byzantine FIFO broadcast, with slight variations in their assumptions. From a practical standpoint, an important variation is between algorithms that are based on digital signatures [MMR97, Rei94], and those based on authenticated links [Bra87, CGR11]. Intuitively, the former class of algorithms trade computation for bandwidth (i.e., tend to fare better in bandwidth-hungry environments), and the latter class of algorithms vice versa.

In our own implementation of the broadcast primitive we use the algorithm based on authenticated links, which appears in several other works, including practical systems [BT85, CGR11, CP02]. We also experiment with a protocol based on digital signatures, but our preliminary results show that this exhibits lower performance than the one based on authenticated links. It would be very interesting to do a thorough investigation on the performance difference between these two versions of Byzantine broadcast, especially for wide-area deployments.

7.5.5 Extensions

We briefly mention here three interesting directions for future work: reconfiguration, tackling the permissionless model, and smart contracts.

In its current form, Astro does not allow a dynamic set of nodes. Typically, changing the set of participants in a distributed system is a delicate operation which has to happen atomically, i.e., participants have to agree on the configuration [LMZ09]. Typically, reconfiguration mechanisms are implemented using consensus within state machine replication systems [SRMJ12, BSA14]. In the context of read-write storage systems, it has been shown that reconfiguration can be done without using consensus under certain failure assumptions [AKMS11]. As described in recent work, a transfer system has consensus number one, similar to read-write storage [GKM+18], so it would be interesting to discover whether token transfer systems can be reconfigured without consensus.

Throughout this thesis we consider the permissioned model. This implies the assumption of an access control mechanism, specifying who is allowed to participate in the system. We assume that this mechanism is external to the system itself. Private protocols, such as Corda [Hea16] or Hyperledger Fabric [ABB+18] rely on such a mechanism. This control mechanism rules out the possibility of Sibyl attacks [Dou02], where a malicious party can take control over a system by using many identities, toppling the one-third assumption on the fraction of Byzantine participants. In case of such an attack, the malicious party can engage

in double-spending, i.e., spend the same tokens more than once.

Decentralized systems for the public, i.e., permissionless, setting are open to the world. They do not have an explicit access control mechanism and allow anyone to join. Systems which fall into this category include Bitcoin [Nak08], Ethereum [Woo15], ByzCoin [KJG+16], Algorand [GHM+17]. To prevent malicious parties from overtaking the system, these systems rely on Sybil-proof techniques, e.g., proof-of-work [Nak08], or proof-of-stake [GHM+17].

To deploy Astro in a public setting, we need to address Sibyl attacks. This is possible using a Sybil-resistant implementation of Byzantine broadcast, while keeping the basic transfer algorithms unchanged (§7.3.2). To the best of our knowledge, there is no such implementation of Byzantine broadcast, however, so we leave this problem open to future research.

Most of the systems we mentioned earlier employ a consensus algorithm [ABB+18, Hea16, Woo15]. Given the powerful building block of consensus, Ethereum [Woo15] for instance generalizes the notion of token transfers to that of arbitrary operations on the system state, known as smart contracts. Our focus in Astro is to obtain an efficient solution for the more narrow problem of decentralized token transfer. We support certain types of contracts that are expressive enough to implement several widely-used design patterns, for instance, tokens, authorizations, or oracles [Btc19, BP17]. Going beyond these, however, requires consensus. We believe an interesting design is possible by merging our efficient solution based on broadcast with a consensus algorithm, employing consensus only for interactions which require coordination among multiple parties.

7.6 Concluding Remarks

In this chapter we studied the problem of implementing a token transfer system efficiently. Since the rise of the Bitcoin cryptocurrency, this problem has garnered significant innovation. Most of the innovation, however, has focused on improving the original solution which Bitcoin proposed, namely, that of using a consensus mechanism to build a total order across all transactions in the system (often called a blockchain). We took a different approach, proposing the Exa abstraction that enables an efficient solution without consensus.

We implemented the Exa abstraction in the Astro transfer system. We also compared Astro with a transfer system based on BFT-Smart, a state-of-the-art SMR system. We observed that Astro provides performance superior to that of the consensus-based solution. In systems of up to 100 replicas, regardless of system size, we observed a throughput improvement ranging from $1.5x$ to $6x$, while achieving a decrease in latency of up to $2x$. Astro also offers improved availability in the presence of failures or asynchrony in the network.

Conclusions

midway on the path
eternity in all directions
stands

— Vlado Škafar [Šk15]

In this dissertation we studied the topic of efficiency in reliable distributed systems. We discussed techniques to mitigate the cost (e.g., in terms of performance, inconsistencies, or inefficiencies) of replication. The goal was to to make replication more transparent to users.

We studied three specific problems.

First, we looked at State Machine Replication (SMR). This is an important technique in distributed systems, owing to its wide applicability and to its ability to ensure strong consistency even in the presence of Byzantine failures. We considered the issue of performance decay due to increasing system size. On this topic, we took steps to further our understanding of this issue by deploying five SMR systems at a large scale and reporting our observations on their decay. Our empirical observations show that chain- or ring-based overlays are more efficient dissemination schemes than leader-centric (i.e., star) schemes.

Overlays based on chain or ring overlays can alleviate performance decay in SMR, though admittedly they do so at the expense of lowered availability. This is because conventional systems based on these dissemination schemes forfeit progress in the presence of asynchrony or faults. We also discussed the design of Carousel, a ring-based system that aims to mitigate performance decay while preserving availability (Part II).

By exploring the chain- and ring-based overlays we hope to emphasize the importance of the dissemination layer, as a deciding factor on the efficiency—and performance decay—of SMR systems. We also hope to motivate further research on other types of overlays, towards understanding and mitigating the problem of performance decay in SMR.

Second, we studied the consistency versus performance tradeoff in replicated systems. We approached this problem from a high-level perspective, and described the Correctables abstraction for supporting efficient access to replicated data. Concretely, this abstraction hides

the complexity of replicated data stacks and allows programmers to access incremental consistency guarantees for any operation, allowing them to balance performance with consistency in their applications (Part III).

The insight behind Correctables is simple: Every consistency model has its own drawbacks (high overhead, performance decay, inconsistencies), so by combining them we hope to go beyond their individual tradeoffs. Our discussion on using Correctables is limited because we focused on certain data types (read-write storage and distributed queues). In the future, we are interested to see how to the idea of incremental consistency guarantees can apply to other problems, e.g., in applications such as those for digital trust, in sharded designs, or in permissionless or other system models.

Third, we turned our attention to the problem of implementing efficient token transfer applications. Traditionally, these applications build on an expensive building block of consensus (i.e, SMR) algorithm. We propose the Exa abstraction that allows eschewing the need for consensus. The Astro system implements this abstraction by building on a broadcast-based primitive with weak ordering guarantees. This primitive is more efficient and tractable than solving consensus (Part IV).

The limitations of a broadcast-based primitive and the Exa abstraction, in the context of digital trust applications, are an open question, especially considering the power of smart contracts. The Exa abstraction supports extension to a sharded system model, though we did not give a full description of how to achieve this. We believe the Exa abstraction applied in a sharded design permits cross-shard coordination without the need for an atomic commit protocol (e.g., two-phase commit). Such a protocol has been in prior work a significant source of overheads and safety issues, so we plan to work on a sharded solution based on Exa in the near future. Beside this extension, in a more broader context, we hope to have motivated a research agenda that focuses more on the needs of applications (and their precise semantics) and less on their building blocks.

A Carousel Supplementary Material

We provide additional details on the Carousel SMR system (Chapter 4), concretely a discussion on the guarantees of this system.

A.1 Safety and Liveness Guarantees for Carousel

Carousel provides the following safety and liveness guarantees.

Safety. We define three safety properties:

1. Only values (i.e., entry ids) that are proposed by replicas can become stable;
2. Per sequence number, only one value can become stable;
3. The same value cannot become stable at two different sequence numbers.

Liveness. Carousel ensures two liveness properties:

1. At any point in time, if there exists a value proposed by a correct replica which is not part of the stable log yet, then eventually one value (possibly another one) will become stable;
2. If there are only a finite number of proposed values, then all of them eventually become stable.

Below, we argue for the correctness of Carousel.

Theorem 1. *Carousel is safe.*

Proof. (1) and (2) follow from the safety properties of FaB [MA06].

(3) This property follows from the fact that correct replicas index values by their identifier *id* (see lines 6 and 19 in Listing 4.1), hence these replicas will not participate in voting or accepting any value that was previously proposed for a different sequence number.

□

Theorem 2. *Carousel is live.*

Proof. (1) Assume there is a value v (entry id) which is not part of the stable log yet. Let sn be the first sequence number for which no value was accepted. If the current sequencer is correct, then it will propose a unique value for sequence number sn (value v, or another value), and the replicas will reach agreement on that value.

If the replicas are not able to eventually reach agreement for sequence number sn, this means that the current sequencer is faulty, and they will trigger a reconfiguration. The new sequencer might be faulty as well, but after enough reconfigurations (at most F), a correct sequencer is chosen. This new sequencer will then propose a unique value for sequence number sn that will become accepted by all (correct) replicas, and accepted to the stable log.

(2) We apply the first liveness property as many times as there are proposed values. Together with safety, this ensures that every operation that needs to be ordered eventually becomes stable.

□

B Astro Supplementary Material

This appendix comprises two sections. First, we discuss the guarantees of the Astro system (Appendix B.1). Second, we provide the pseudocode for the Byzantine FIFO broadcast primitive which is important in implementing Astro (Appendix B.2).

B.1 Safety and Liveness Guarantees for Astro

Our Astro algorithm (§7.3.2) ensures the following properties.

Property 1. ***Liveness.*** *If a correct node* k *initiates a transfer and has sufficient balance, then eventually all correct nodes in the committee of* k *apply that transfer.*

Property 2. ***Safety.*** *If a correct node* i *applies transfer* $T1$ *before applying* $T2$*, then no correct node applies* $T2$ *before* $T1$*, where both transfer* $T1$ *and* $T2$ *have the same spender account* k.

We start by proving a helpful lemma.

Lemma 1. *Consider any two correct nodes* i *and* j *belonging to the committee of some account* k*. If the local copy of* hist[k] *at node* i *has value* h *when sequence number* sn[k] *is* n*, then* hist[k] *at node* j *also has value* h *when sequence number* sn[k] *is* n *at node* j.

Proof. We note that each correct node updates its local variables *hist[k]* and *sn[k]* in tandem (lines 30, 31 in Listing 7.5). This update is performed only when applying a transfer outgoing from account k. This happens after node k sends a transfer message with the details of the transfer. We make two key observations.

(i) By the properties of Byzantine FIFO broadcast, no two correct nodes in the committee of k deliver two different messages for the same sequence number n. In other words, these nodes observe the same transfer details for a given sequence number (variables t and D on line 30).

(ii) Transfers outgoing from the same account k are always applied in increasing order of their corresponding sequence number n (see line 49). By *applying* a transfer we mean executing

the algorithm in Listing 7.5 (lines 28–34).

Since any two correct nodes i and j in the committee of k perform (i) the same updates to their local variable *hist[k]* and (ii) in the same order as given by *sn[k]*, it follows that for the same value of *sn[k]*, *hist[k]* has identical value at nodes i and j. □

Theorem 3. *Astro ensures liveness.*

Proof. We start from the observation that the balance of account k at node k is sufficient when this node starts its transfer. This means that the balance check passes (line 17), so node k broadcasts a transfer message m with the transfer details (line 20). By the liveness property of Byzantine FIFO broadcast, all correct nodes in the committee of k eventually deliver message m. To ensure liveness and guarantee that these nodes apply the transfer of k, message m must fulfill several validation criteria (lines 48–51) described algorithmically in the *valid* function (Listing 7.7). We discuss each validation criteria in turn.

Since node k is correct, then message m is well-formed, meaning that m contains the identity k of this node (the validation on line 48 passes) as well as the correct monotonically increasing sequence number n (validation on line 49).

The set of dependencies for this transfer are well-formed. Node k assembled this set by doing four verification steps. (i) All proofs are for transfers incoming to account k (line 38). (ii), (iii) For a given transfer outgoing from some account a, all proofs are sent and signed by nodes that are part of the committee of account a (lines 39 and 40). (iv) Each dependency has at least *faultThreshold + 1* proofs (line 44). Any correct node that delivers message m does the same four verification steps via function *checkDeps*: line 56 for verification (i); lines 59, 60 for (ii), (iii), respectively; and line 55 for step (iv). Since these four steps pass, then *checkDeps* returns true and all dependency checks are fulfilled (validation of dependencies on line 50 passes).

The last validation criteria for message m regards the balance check. Recall that the balance check correctly passed at node k (before this node broadcast transfer message m). Node k performed this check against *hist[k]* at a certain sequence number n. By Lemma 1, we know that the variable *hist[k]* is identical at two different correct nodes from committee of k for the same sequence number *sn[k]* = n. Hence, the balance check also passes at the other correct nodes in committee of k (validation on line 51 passes).

Since all the validation criteria in the *valid* function pass, then each correct node in committee of k eventually applies this transfer. □

Theorem 4. *Astro ensures safety.*

Proof. Observe that if any correct node applies some transfer with spender account k, then such a node is part of the committee of account k. Both transfers $T1$ and $T2$ have the same

spender k. We denote by *n1* the sequence number of transfer $T1$, and by *n2* the sequence number of transfer $T2$.

Assume by contradiction that there exist two correct nodes i and j which apply transfers $T1$ and $T2$ in different order. Specifically, node i applies $T1$ before $T2$ (and node j in the opposite order). Denote by h the local variable *hist[k]* at node i for sequence number *n1*. This h value is derived from all transfer which node i previously applied involving account k, including dependencies, but h does not include transfer $T2$ (because this was not yet applied at sequence number *n1* at node i).

Now consider node j. By Lemma 1, the local variable *hist[k]* at node j for sequence number *n1* must also have value h. But recall that we assume node j applies $T2$ before $T1$. This means that *hist[k]* at node j must include transfer $T2$, a contradiction. □

B.2 Byzantine FIFO Broadcast Algorithm

In Listing B.1 we sketch the pseudocode for implementing Byzantine FIFO broadcast. Recall that this algorithm proceeds in three phases, as we described earlier (§7.3.3). We now reproduce this description to help understand the pseudocode below. Consider a full replication model, where messages are broadcast to all of the system nodes. As the underlying communication layer, we assume authenticated links, e.g., based on Message Authentication Codes (MACs). Whenever we use the function *sendToAll*, we mean to say that the local node sends a certain message to all nodes in the system, including itself, via authenticated links (lines 4, 10, 17, 24).

Recall that the Byzantine FIFO algorithm we use comprises three phases, identified by the type of protocol messages of that phase, as follows.

1. **Send—**To initiate a broadcast, any correct node i simply attaches a sequence number s to its message m (line 3), and then sends this tuple *{m, s}* to all nodes (line 4).
2. **Echo—**The first time a node j receives the tuple *(m,s)* from i, it sends an echo message for this tuple to all nodes in the system (line 10).
3. **Ready—**In this last phase of the algorithm, every node waits to gather a Byzantine quorum of echo messages for tuple *(m,s)* (line 14) and then sends a ready message (line 17). Alternatively, a node may send a ready message after observing $F+1$ ready messages (lines 21 and 24). Any node may deliver message m after gathering $2F+1$ matching ready messages for m and the given sequence number (line 29).

```
1  // Called at replica p to initiate a broadcast.
2  func broadcast(m):
3    m := ⟨SEND, myTS[p] + 1, m⟩
4    sendToAll(m)  // Send to all replicas.

6  // Handle the receiving at replica p of protocol message m from replica q.
7  callback receive(q, m = ⟨SEND, ts, m⟩):  // Handler for SEND messages.
8    if echoSent[q, ts] == false:
9      echoSent[q, ts] := true
10     sendToAll(⟨ECHO, q, ts, m⟩)

12 callback receive(q, m = ⟨ECHO, r, ts, m⟩):  // Handler for ECHO messages.
13   echoes[r, ts, m] += q
14   if (|echoes[r, ts, m]| ≥ 2F+1) &&
15       (readySent[r, ts, m] == false):
16     readySent[r, ts] := true
17     sendToAll(⟨READY, r, ts, m⟩)

19 callback receive(q, m = ⟨READY, r, ts, m⟩):  // Handler for READY messages.
20   readys[r, ts, m] += q
21   if (|readys[r, ts, m]| ≥ F+1) &&
22         (readySent[r, ts, m] == false):
23     readySent[r, ts, m] := true
24     sendToAll(⟨READY, r, ts, m⟩)
25   if (|readys[r, ts, m]| ≥ 2F+1) &&
26         (delivered[r, ts, m] == false) &&
27         (ts == myTS[r] + 1):
28     delivered[r, ts, m] := true
29     trigger deliver(r, m)
30     myTS[r] += 1
```

Listing B.1 – Pseudocode for Byzantine FIFO broadcast.

Bibliography

[AAC+05] Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. *ACM SIGOPS Operating Systems Review*, 39(5):45–58, 2005.

[Aba12] Daniel J Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, 45(2):37–42, 2012.

[ABB+18] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *EuroSys*, 2018.

[ABK+15] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to Adopting Stronger Consistency at Scale. In *HotOS XV*, 2015.

[ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[ADP18] Eduardo Alchieri, Fernando Dotti, and Fernando Pedone. Early Scheduling in Parallel State Machine Replication. In *ACM SoCC*, 2018.

[ADW10] Atul Adya, John Dunagan, and Alec Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *NSDI*, 2010.

[AEMGG+05] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine Fault-tolerant Services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[AFK+09] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP*, 2009.

[AGK+15] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The Next 700 BFT Protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4):12:1–12:45, 2015.

[AGM+17] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. *arXiv/cs.DC*, 1712.01367, 2017. http://arxiv.org/abs/1712.01367.

[AGMS18] Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. State Machine Replication is More Expensive Than Consensus. In *DISC*, 2018.

[AGTK15] Kyoungho An, Aniruddha Gokhale, Sumant Tambe, and Takayuki Kuroda. Wide area network-scale discovery and data dissemination in data-centric publish/-subscribe systems. In *Middleware*, 2015.

[AKMS11] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *Journal of the ACM (JACM)*, 58(2):7:1–7:32, 2011.

[ALvRV13] Hussam Abu-Libdeh, Robbert van Renesse, and Ymir Vigfusson. Leveraging Sharding in the Design of Scalable Replication Protocols. In *ACM SoCC*, 2013.

[AMN+18] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus. In *OPODIS*, 2018.

[AMS+07] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP*, 2007.

[ANRST05] Yair Amir, Cristina Nita-Rotaru, S Stanton, and Gene Tsudik. Secure spread: An integrated architecture for secure group communication. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2(3):248–261, 2005.

[App18] AppEngine. Google Cloud Platform, Accessed January, 2018. https://appengine.google.com/.

[Arm13] Joe Armstrong. *Programming Erlang: software for a concurrent world*. Pragmatic Bookshelf, 2013.

[AS84] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1984.

[AS85] Bowen Alpern and Fred B Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.

[AW94] Hagit Attiya and Jennifer L. Welch. Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, 1994.

[BBC+11] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[BCvR09] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a cloud computing research agenda. *ACM SIGACT News*, 40(2):68–80, 2009.

[BDF+13] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *VLDB*, 7(3):181–192, 2013.

[BFF+15] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *SIGMOD*, 2015.

[BFG+12] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *ACM SoCC*, 2012.

[Bft19a] Bft-SMaRt home page, Accessed March, 2019. http://bft-smart.github.io/library/.

[Bft19b] bft-smart/library, commit 3af266d4 dated Jul 12, 2016, Accessed March, 2019. https://github.com/bft-smart/library/commit/3af266d4.

[BJS11] Diogo Becker, Flavio Junqueira, and Marco Serafini. Leader election for replicated services using application scores. In *Middleware*, 2011.

[BMSS12] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. S-Paxos: Offloading the Leader for High Throughput State Machine Replication. In *SRDS*, 2012.

[BP16] Carlos Eduardo Bezerra and Fernando Pedone. Strong Consistency at Scale. *IEEE Data Engineering Bulletin*, 2016.

[BP17] Massimo Bartoletti and Livio Pompianu. An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns. In *Financial Cryptography and Data Security*, pages 494–509. Springer International Publishing, 2017.

[BPR14] Carlos Eduardo Bezerra, Fernando Pedone, and Robbert Van Renesse. Scalable State-Machine Replication. In *IEEE/IFIP DSN*, 2014.

[BR92] B. R. Badrinath and Krithi Ramamritham. Semantics-based Concurrency Control: Beyond Commutativity. *ACM Transactions on Database Systems (TODS)*, 17(1):163–199, 1992.

[Bra87] Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[Bre12] Eric Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[BSA14] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with BFT-SMaRt. In *IEEE/IFIP DSN*, 2014.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[Btc19] Contract - bitcoin wiki, Accessed March, 2019. https://en.bitcoin.it/wiki/Contract.

[Bur06] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[But15] Vitalik Buterin. Ethereum / Wiki / Standardized Contract APIs, 2015. https://github.com/ethereum/wiki/wiki/Standardized_Contract_APIs.

[BVF+12] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *VLDB*, 5(8):776–787, 2012.

[Cas01] Miguel Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, Massachusetts Institute of Technology. Department of Electrical Engineering and Computer Science, 2001.

[CDE+13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3), 2013.

[CDE+16] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed E. Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains - (A position paper). In *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, pages 106–125, 2016.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2), 2008.

[CFJS12] Miguel Correia, Daniel Gómez Ferro, Flavio P Junqueira, and Marco Serafini. Practical hardening of crash-tolerant systems. In *USENIX ATC*, 2012.

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *PODC*, 2007.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[CKZ+15] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *ACM Transactions on Computer Systems (TOCS)*, 32(4):10:1–10:47, 2015.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[Cla88] David D. Clark. The Design Philosophy of the DARPA Internet Protocols. *ACM SIGCOMM Computer Communication Review*, 18(4):106–114, 1988.

[CML+06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *OSDI*, 2006.

[CMSK07] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory. In *SOSP*, 2007.

[Cou19] Expert Panel (Forbes Technology Council). 10 Tech Industry Experts Predict The Next 'Blockchain Wave', Feb 13 2019.

[CP02] Christian Cachin and Jonathan A. Poritz. Secure Intrusion-tolerant Replication on the Internet. In *IEEE/IFIP DSN*, 2002.

[CPVK16] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, and Vignesh Kalyanaraman. Blockchain technology: Beyond bitcoin. *Applied Innovation Review*, 2(6):6–19, 2016.

[CRS+08] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *VLDB*, 1(2):1277–1288, 2008.

[CST$^+$10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *ACM SoCC*, 2010.

[CV17] Christian Cachin and Marko Vukolic. Blockchain Consensus Protocols in the Wild (Keynote Talk). In *DISC*, 2017.

[CWA$^+$09] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *NSDI*, 2009.

[DB13] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[DHJ$^+$07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[DIRZ14] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *ACM SoCC*, 2014.

[DNN$^+$15] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *SOSP*, 2015.

[Dou02] John R Douceur. The Sybil Attack. In *IPTPS*, 2002.

[DRZ18] Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: Asynchronous BFT Made Practical. In *CCS*, 2018.

[Ec218] Amazon EC2 instance types. https://aws.amazon.com/ec2/instance-types/, Accessed September, 2018.

[EGSVR16] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. In *NSDI*, 2016.

[Eri13] Marius Eriksen. Your server as a function. In *PLOS*, 2013.

[Etc19a] CoreOS/etcd, Accessed March, 2019. https://coreos.com/etcd/.

[Etc19b] coreos/etcd release version 2.3.7 dated June 17, 2016, Accessed March, 2019. https://github.com/coreos/etcd/releases/tag/v2.3.7.

[Etc19c] etcd production users, Accessed March, 2019. https://github.com/coreos/etcd/blob/master/Documentation/production-users.md.

[Etc19d] etcd/etcdserver/raft.go, Accessed March, 2019. https://github.com/coreos/etcd/blob/master/etcdserver/raft.go#L48.

[Fid88] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. of the 11th Australian Computer Science Conference (ACSC'88)*, 1988.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, April 1985.

[Fou12] Camille Fournier. Running zookeeper across regions, 2012. http://www.elidedbranches.com/2012/12/building-global-highly-available.html.

[FP18] Enrique Fynn and Fernando Pedone. Challenges and Pitfalls of Partitioning Blockchains. *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, Jun 2018.

[Fug15] Hans Fugal. Futures for C++11 at Facebook, 2015. https://code.facebook.com/posts/1661982097368498.

[GAG+19] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu. Harmony: a Scalable and Decentralized Trust Infrastructure. In *IEEE/IFIP DSN*, 2019.

[GBKA11] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable Consistency in Scatter. In *SOSP*, 2011.

[GGG+18] Alain Girault, Gregor Gössler, Rachid Guerraoui, Jad Hamza, and Dragos-Adrian Seredinschi. Monotonic Prefix Consistency in Distributed Systems. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, pages 41–57. Springer, Cham, 2018.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP*, 2003.

[GHM+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP*, 2017.

[GHSV19] Rachid Guerraoui, Jad Hamza, Dragos-Adrian Seredinschi, and Marko Vukolić. On the non-scalability of state machine replication. *Under submission*, 2019.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *SOSP*, 1979.

[GKM+18] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. AT2: Asynchronous Trustworthy Transfers. *arXiv/cs.DC*, 1812.10844, 2018. http://arxiv.org/abs/1812.10844.

[GKPS16] Rachid Guerraoui, Anne-Marie Kermarrec, Matej Pavlovic, and Dragos-Adrian Seredinschi. Atum : Scalable Group Communication Using Volatile Groups. In *Middleware*, 2016.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), 2002.

[GLPQ10] Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Vivien Quéma. Throughput Optimal Total Order Broadcast for Cluster Environments. *ACM Transactions on Computer Systems (TOCS)*, 28(2):5:1–5:32, 2010.

[GPS16] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental Consistency Guarantees for Replicated Objects. In *OSDI*, 2016.

[Gua19] google/guava wiki: ListenableFutureExplained, Accessed March, 2019. https://github.com/google/guava/wiki/ListenableFutureExplained.

[Gup16] Saurabh Gupta. A Non-Consensus Based Decentralized Financial Transaction Processing Model with Support for Efficient Auditing. Master's thesis, Arizona State University, USA, 2016.

[Ham09] James Hamilton. The cost of latency. http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/, 2009.

[HC09] P. Helland and D. Campbell. Building on quicksand. In *CIDR*, 2009.

[Hea16] Mike Hearn. Corda: A distributed ledger. *Corda Technical White Paper*, 2016.

[Her90] Maurice Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM Transactions on Database Systems (TODS)*, 15(1):96–124, 1990.

[Her91] Maurice Herlihy. Wait-free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[HK11] C. Hale and R. Kennedy. Using Riak at Yammer, March 2011. http://dl.dropbox.com/u/2744222/2011-03-22_Riak-At-Yammer.pdf.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[HMW] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series: Consensus system. Rev.1. https://dfinity.org/tech.

[HW90] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3), 1990.

[Ins19] C++ Futures at Instagram, Accessed March, 2019. https://instagram-engineering.com/c-futures-at-instagram-9628ff634f49.

[ISAV16] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *NSDI*, 2016.

[JMPSP17] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: High-throughput atomic broadcast. *The Computer Journal*, 60(6):866–882, 2017.

[JRS11] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *IEEE/IFIP DSN*, 2011.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM SIGOPS Operating Systems Review*, 41(6), 2007.

[Kem17] M. Kempe. The Land Registry in the blockchain–testbed. *A development project with Lantmäteriet, Landshypotek Bank, SBAB, Telia company, ChromaWay and Kairos Future*, 2017.

[KHAK09] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency Rationing in the Cloud : Pay only when it matters. *VLDB*, 2(1):253–264, 2009.

[KJG+16] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *USENIX Security*, 2016.

[KJW+18] Kolbeinn Karlsson, Weitao Jiang, Stephen Wicker, Danny Adams, Edwin Ma, Robbert van Renesse, and Hakim Weatherspoon. Vegvisir: A Partition-Tolerant Blockchain for the Internet-of-Things. In *ICDCS*, 2018.

[KKJG+17] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, and Bryan Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger. *IACR Cryptology ePrint Archive*, 2017, 2017.

[Kne12] Nikola Knezevic. *A High-Throughput Byzantine Fault-Tolerant Protocol*. PhD thesis, EPFL, 2012.

[Kor10] Yehuda Koren. Collaborative filtering with temporal dynamics. *Communications of the ACM*, 53(4):89–97, 2010.

[KPF+13] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data Center Consistency. In *EuroSys*, 2013.

[KWQ+12] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about Eve: execute-verify replication for multi-core servers. In *OSDI*, 2012.

[Lam77] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Lam83] Leslie Lamport. The Weak Byzantine Generals Problem. *Journal of the ACM (JACM)*, 30(3):668–676, jul 1983.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), 1998.

[Lam03] Leslie Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, 2003.

[LCC+15] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming. In *MobiSys*, 2015.

[LDR08] John R Lange, Peter A Dinda, and Samuel Rossoff. Experiences with Client-based Speculative Remote Display. In *USENIX ATC*, 2008.

[LFKA11] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *SOSP*, 2011.

[LFKA13] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI*, 2013.

[Lib18] LibPaxos3. https://bitbucket.org/sciascid/libpaxos, Accessed September, 2018.

[LLC+14] Cheng Li, Joao Leitao, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *USENIX ATC*, 2014.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[LMNR15] Ding Li, James Mickens, Suman Nath, and Lenin Ravindranath. Domino: Understanding Wide-Area, Asynchronous Event Causality in Web Applications. In *ACM SoCC*, 2015.

[LMZ09] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *PODC*, 2009.

[LPC+12] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, 2012.

[LPK+15] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *SOSP*, 2015.

[LS88] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI*, 1988.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[LVA+15] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential Consistency : Measuring and Understanding Consistency at Facebook. In *SOSP*, 2015.

[LVC+16] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. XFT: Practical Fault Tolerance Beyond Crashes. In *OSDI*, 2016.

[MA06] J. P. Martin and L. Alvisi. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 3(3):202–215, July 2006.

[MAD11] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. *University of Texas at Austin*, Technical Report TR-11-22, 2011.

[Mal18] Alexey Malanov. Cryptocurrency Threat Predictions for 2019. KasperskyLab, November, 2018.

[Mat88] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proc. Workshop on Parallel and Distributed Algorithms*, 1988.

[MEHL10] James Mickens, Jeremy Elson, Jon Howell, and Jay Lorch. Crom: Faster Web Browsing Using Speculative Execution. In *NSDI*, 2010.

[Mei12] Erik Meijer. Your mouse is a database. *Communications of the ACM*, 55(5):66–73, 2012.

[MJM08] Y. Mao, F. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. In *OSDI*, 2008.

[MMN+04] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, 2004.

[MMR97] Dahlia Malkhi, Michael Merritt, and Ohad Rodeh. Secure Reliable Multicast Protocols in a WAN. In *ICDCS*, 1997.

[MPS93] Shivakant Mishra, Larry L Peterson, and Richard D Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1(2):87, 1993.

[MR97] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. In *STOC*, volume 97, pages 569–578, 1997.

[MRWW01] Dahlia Malkhi, Michael K. Reiter, Avishai Wool, and Rebecca N. Wright. Probabilistic quorum systems. *Information and Computation*, 170(2):184–206, 2001.

[MXC+16] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *CCS*, 2016.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[NCF05] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative Execution in a Distributed File System. In *SOSP*, 2005.

[Net19a] Distributed queue. netflix/curator, Accessed March, 2019. https://github.com/Netflix/curator/wiki/Distributed-Queue.

[Net19b] ProgressivePromise (Netty 4.0 API), Accessed March, 2019. http://netty.io/4.0/api/io/netty/util/concurrent/ProgressivePromise.html.

[NFG+13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling Memcache at Facebook. In *NSDI*, 2013.

[Ong14] Diego Ongaro. *Consensus: Bridging theory and practice.* PhD thesis, Stanford University, 2014.

[PAA+15] Dorian Perkins, Nitin Agrawal, Akshat Aranya, Curtis Yu, Younghwan Go, Harsha V. Madhyastha, and Cristian Ungureanu. Simba: Tunable End-to-end Data Consistency for Mobile Apps. In *EuroSys*, 2015.

[PH15] Marius Poke and Torsten Hoefler. DARE: High-Performance State Machine Replication on RDMA Networks. In *HPDC*, 2015.

[PKFF14] Gene Pang, Tim Kraska, Michael J. Franklin, and Alan Fekete. PLANET: Making Progress with Commit Processing in Unpredictable Environments. In *SIGMOD*, 2014.

[PLL+15] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Paiva Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *EuroSys*, 2015.

[PP13] Ricardo Padilha and Fernando Pedone. Augustus: scalable and robust storage for cloud applications. In *EuroSys*, 2013.

[PS02] Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

[QAB+13] Lin Qiao, Aditya Auradar, Chris Beaver, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jgadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Kapil Surlaker, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, David Zhang, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, and Zhen Zhang. On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform. In *SIGMOD*, 2013.

[Red16] reddit/r2/r2/lib/comment_tree.py:308, Accessed March, 2016. Source https://github.com/reddit/reddit.

[Rei94] Michael K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *CCS*, 1994.

[Ria19] RIAK KV, Accessed March, 2019. http://basho.com/products/riak-kv/.

[RJKL11] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

[SBV18] Joao Sousa, Alysson Bessani, and Marko Vukolic. A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform. In *IEEE/IFIP DSN*, 2018.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[Sch14] Peter Schulle. Manhattan, distributed database for Twitter scale. http://tiny.cc/twitmanhattan, 2014.

[Sch15] Malte Schwarzkopf. *Operating system support for warehouse-scale computing*. PhD thesis, University of Cambridge Computer Laboratory, 2015.

[Sim19] Amazon simpledb, Accessed March, 2019. https://aws.amazon.com/simpledb/.

[Sky] Skyscanner. http://www.skyscanner.ch.

[Sof] Softlayer. http://www.softlayer.com/our-platform.

[SPBZ11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*. Springer, 2011.

[SRMJ12] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Paiva Junqueira. Dynamic Reconfiguration of Primary/Backup Clusters. In *USENIX ATC*, 2012.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[SS12] Nuno Santos and André Schiper. Tuning paxos for high-throughput with batching and pipelining. In *International Conference on Distributed Computing and Networking*, pages 153–167. Springer, 2012.

[Sza97] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[T+11] Ole Tange et al. GNU parallel - The Command-Line Power Tool. *The USENIX Magazine*, 36(1):42–47, 2011.

[Ten19] TendermintCore, Accessed March, 2019. https://github.com/tendermint/tendermint.

[TPK+13] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.

[TTP+95] D. B. Terry, M. M. Theimer, Karin Petersen, a. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review*, 29(5):172–182, 1995.

[Twe19] Followthehashtag / 170,000 Apple tweets, Accessed March, 2019. http://followthehashtag.com/datasets/170000-apple-tweets-free-twitter-dataset/.

[Twi19] Twissandra, Accessed March, 2019. https://github.com/twissandra/twissandra/.

[Ups] Upstart: event-based init deamon. http://upstart.ubuntu.com.

[VB15] Fabian Vogelsteller and Vitalik Buterin. EIP 20: ERC-20 Token Standard, 2015. https://eips.ethereum.org/EIPS/eip-20.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[VRHS12] Robbert Van Renesse, Chi Ho, and Nicolas Schiper. Byzantine chain replication. In *OPODIS*, 2012.

[VRS04] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[VRSS15] Robbert Van Renesse, Nicolas Schiper, and Fred B Schneider. Vive la différence: Paxos vs. Viewstamped Replication vs. ZAB. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 12(4):472–484, 2015.

[Vuk15] Marko Vukolić. The Quest for Scalable Blockchain Fabric: Proof-of-work vs. BFT Replication. In *International Workshop on Open Problems in Network Security*, pages 112–125. Springer, 2015.

[Wal00] Arthur Waley. *The Analects of Confucius*. Everyman's Library (Alfred A. Knopf), 2000.

[Wal13] Arthur Waley. *The way and its power: A study of the Tao Te Ching and its place in Chinese thought*. Routledge, 2013.

[WBP+13] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. SPANStore: Cost-Effective Geo-Replicated Storage Spanning Multiple Cloud Services. In *SOSP*, 2013.

[WCF11] Benjamin Wester, Peter M Chen, and Jason Flinn. Operating System Support for Application-Specific Speculation. In *EuroSys*, 2011.

[WCN+09] Benjamin Wester, James A Cowling, Edmund B Nightingale, Peter M Chen, Jason Flinn, and Barbara Liskov. Tolerating latency in replicated state machines through client speculation. In *NSDI*, pages 245–260, 2009.

[WFZ+11] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. In *CIDR*, 2011.

[WKR+13] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the Salus scalable block store. In *NSDI*, 2013.

[Woo15] Gavin Wood. Ethereum: A secure decentralized generalized transaction ledger. White paper, 2015.

[XSK+14] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining ACID and BASE in a distributed database. In *OSDI*, 2014.

[YMR+18] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus in the Lens of Blockchain. *arXiv/cs.DC*, 1803.05069, 2018. http://arxiv.org/abs/1803.05069.

[YMV+03] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *SOSP*, 2003.

[YV00] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *OSDI*, 2000.

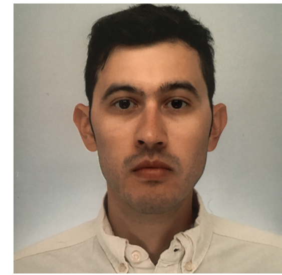
[Zoo14] ZooKeeper (Ubuntu package), high-performance coordination service for distributed applications, 2014. http://packages.ubuntu.com/trusty/zookeeper.

[Zoo19] ZooKeeper Recipes and Solutions, Accessed March, 2019. http://tiny.cc/zkqueues.

[ZSS+15] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building Consistent Transactions with Inconsistent Replication. In *SOSP*, 2015.

[Šk15] Vlado Škafar. *Krogi=En=Circles*. Ljubljana : Kinetik, 2015.

DRAGOS-ADRIAN SEREDINSCHI



Distributed Computing Lab (DCL)
School of Computer and Communication Sciences
EPFL

adi.seredinschi@protonmail.com ✉
+41.78.741.8707 ☎

QUALIFICATIONS & INTERESTS

I am interested, broadly speaking, in efficient designs for distributed systems. My focus is both on understanding their foundations from a theoretical perspective, as well as implementing these systems efficiently. In particular, I work on replication, consistency, and agreement protocols.

EDUCATION

EPFL, *Lausanne, Switzerland* — 2013 – 2019
Ph.D. in Computer Science
Advisor: Rachid Guerraoui
Thesis: System Support for Efficient Replication in Distributed Systems

Babeş-Bolyai University, *Cluj-Napoca, Romania* — 2011 – 2013
M.Sc. in Computer Science
Advisor: Sterca Adrian
Thesis: Network Layer Interface Aggregation for IP Tunneling

B.Sc. in Computer Science — 2008 – 2011
Advisor: Sterca Adrian
Thesis: 15 Years Later: Still Smashing the Stack for Fun and Profit

Université d'Orléans, *Orléans, France* — 01/2012 – 07/2012
M.Sc. Semester exchange with Erasmus scholarship
Advisor: Ioan Todinca
Specialization: Distributed Systems (Répartition et Aide à la Décision)

PROFESSIONAL EXPERIENCE

VMware Research Group, *Palo Alto, CA, USA* — 06/2017 – 09/2017
Research Intern
Host: Dahlia Malkhi
Topic: Scalable Decentralized Trust Infrastructure for Blockchains

IBM Research – Zurich, *Rüschlikon, Switzerland* — 06/2016 – 09/2016
Research Intern
Host: Marko Vukolić
Topic: Understanding the Non-Scalability of Consensus Protocols

Distributed Computing Lab, EPFL, *Lausanne, Switzerland* — 07/2013 – 09/2013
Research Intern
Host: Rachid Guerraoui
Topic: Data Streaming Application for Large-Scale BFT Systems

Ocedo, *Cluj-Napoca, Romania* 04/2013 – 06/2013
Software Engineer
Business type: Computer Networking (Software Defined WAN) Startup
Ocedo was thereafter acquired by Riverbed Technology
Project: Data Authentication & Encryption for a Distributed Logging Service

Sophos, *Cluj-Napoca, Romania* 07/2011 – 03/2013
Software Engineer
Business type: Computer Security
Designed and implemented a three-tier cloud-based application (several front-end web services, a middleware data broker, and a MySQL back-end)

Evozon Systems, *Cluj-Napoca, Romania* 07/2010 – 04/2011
Junior Software Developer
Business type: Software Development & Outsourcing
Activities: complete software development life cycle, Scrum-based project management, database setup and management

TEACHING EXPERIENCE

Teaching Assistant, Distributed Algorithms 2015, 2016, 2017, 2018
M.Sc. course dedicated to introducing fundamental result, abstractions, and algorithms for distributed systems. I have taught several lessons, supervised all exercise sessions, as well as participated in designing and grading exams.

Project Supervisor, Distributed Algorithms 2016, 2018
Project assignment involving the implementation of a distributed storage with various levels of consistency

Teaching Assistant, Programming I 2014
B.Sc. course that uses C++ to introduce students to programming

Teaching Assistant, Programming II 2017
B.Sc. course that uses C++ to introduce students to programming

Teaching Assistant, Introduction to Programming 2015, 2016
B.Sc. course that uses C++ to introduce students to programming

SUPERVISED STUDENTS & MENTORING

Athanasios Xygkis Ph.D candidate, semester project 12/2018 – 04/2019
Sharding an Asynchronous Token Transfer Protocol

Ye Wang Ph.D candidate, semester project 12/2018 – 04/2019
Dynamic Token Transfer Protocols without Consensus

Guillaume Vizier M.Sc. student, semester project 09/2018 – 01/2019
Reconfiguration of BFT Algorithms

Vincenzo Bazzucchi M.Sc. student, semester project 09/2018 – 01/2019
The impact of leader election strategy on Paxos-based state machine replication
Co-supervised with Karolos Antoniadis

Matteo Monti Ph.D candidate 11/2017 – 06/2018
Scalable Byzantine Broadcast

Bogdan Suvar M.Sc. thesis 10/2017 – 07/2018
Evaluating RPC frameworks for improving Raft throughput

Jiacheng Xu M.Sc. student, semester projects 09/2017 – 06/2018
1. *Reconfiguration in a Ring Replication Protocol*
2. *Security and Practical Aspects in the Carousel SMR Protocol*

Agapiou Stylianos. M.Sc. student, semester project 02/2017 – 06/2017
A Group Membership Service based on etcd/raft

Bertrand Christophe. M.Sc. student, semester project 02/2017 – 06/2017
Group-based Chain Replication using Raft

Najdenova Iva. M.Sc. student, semester project 02/2017 – 06/2017
Implementation of Group Membership using ZooKeeper

CONFERENCE TALKS & PAPERS

The Consensus Number of a Cryptocurrency 2019
Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi
In Proceedings of the 38th Annual ACM Symposium on Principles of Distributed Computing (PODC 2019)

Harmony: a Scalable and Decentralized Trust Infrastructure 2019
Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu
In Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN 2019)

State Machine Replication Is More Expensive Than Consensus 2018
Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, Dragos-Adrian Seredinschi
In 32nd International Symposium on Distributed Computing (DISC 18)

Monotonic Prefix Consistency in Distributed Systems 2018
Alain Girault, Gregor Goessler, Rachid Guerraoui, Jad Hamza, D.A. Seredinschi
In 13th International Federated Conference on Distributed Computing Techniques

Blockchain Protocols: The Adversary is in the Details 2018
Rachid Guerraoui, Matej Pavlovic, Dragos-Adrian Seredinschi
In 1st Symposium on Foundations and Applications of Blockchain

Incremental Consistency Guarantees for Replicated Objects 2016
R. Guerraoui, M. Pavlovic, D.A. Seredinschi
In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)

Atum: Scalable Group Communication Using Volatile Groups 2016
R. Guerraoui, A.M. Kermarrec, M. Pavlovic, D.A. Seredinschi
In ACM/IFIP/USENIX Middleware

Network Interface Aggregation for IP Tunneling 2013
D. A. Seredinschi, A. Sterca
In Knowledge Engineering: Principles and Techniques Conference (KEPT)

15 years later: Still Smashing the Stack for Fun and Profit 2011
D.A. Seredinschi
In Scientific Communications of the Faculty of Mathematics and Computer Science in UBB and the Faculty of Automation and Computer Science in TU, Cluj-Napoca

JOURNAL ARTICLES

Trade-offs in Replicated Systems 2016
R. Guerraoui, M. Pavlovic, D. A. Seredinschi
IEEE Data Engineering Bulletin, 39(1), March, 2016

Network Interface Aggregation for IP Tunneling 2013
D. A. Seredinschi, A. Sterca
Studia Universitatis Babeş-Bolyai, Series Informatica, LVIII(3)

Enhancing The Stack Smashing Protection in The GCC 2011
D. A. Seredinschi, A. Sterca
Studia Universitatis Babeş-Bolyai, Series Informatica, LVI(4)

HONORS

Doctoral School Fellowship. EPFL 2013 – 2014
ERASMUS Master Student Scholarship. UniversitÃľ d'OrlÃľans 2012
Study Scholarship – Master. Babeş-Bolyai University 2011 – 2013
Study Scholarship – Bachelor. Babeş-Bolyai University 2009 – 2011

PERSONAL SKILLS AND COMPETENCES

Mother tongue: **Romanian**
Foreign languages:
English—Level C1 of the Common European Framework for Languages
French—Level B2 of the Common European Framework for Languages

Social and organisational skills:

- Highly adaptive
- Thinking outside (and inside) the box
- Sense of personal and social responsibility
- Reading—basically anything

Ce document a été imprimé au Centre d’impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.