EPFL

Thèse n° 9909

Reliable Microsecond-Scale Distributed Computing

Présentée le 6 octobre 2023

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

pour l'obtention du grade de Docteur ès Sciences

par

Athanasios XYGKIS

Acceptée sur proposition du jury

Prof. B. Falsafi, président du jury
Prof. R. Guerraoui, directeur de thèse
Prof. G. Alonso, rapporteur
Prof. C. Kozyrakis, rapporteur
Prof. E. Bugnion, rapporteur

École polytechnique fédérale de Lausanne

2023

If I have seen further
it is by standing on the shoulders of Giants.
— Isaac Newton (1642–1727)

To my brother Nicholaos

Acknowledgements

First and foremost I am grateful to my advisor, Prof. Rachid Guerraoui. Thank you for the support, guidance, and constant patience throughout this challenging yet rewarding journey. Through our conversations, you were always keeping me motivated to chase novel research pathways, you taught me how to distill the essence of the problem, and how to pinpoint the fundamentals. Thank you for ensuring the perfect conditions to conduct my research, creating a stress-free environment, where work becomes fun and creativity flourishes.

I am grateful to the members of my PhD oral examination jury: Prof. Edouard Bugnion, Prof. Gustavo Alonso, and Prof. Christos Kozyrakis, all of which I respect immensely. Their questions, comments, and suggestions helped me get a crystal clear understanding of the limitations and possible extensions of the work in this dissertation. I am also thankful to Prof. Babak Falsafi for presiding over my oral examination.

I am immensely grateful to my collaborators throughout these years starting with Dr. Marcos K. Aguilera, whose craftsmanship as a researcher never stops to inspire. I thank my early collaborators, Prof. Petr Kuznetsov, Dr. Yvonne-Anne Pignolet, Dr. Adrian-Dragos Seredinschi, Dr. Virendra Marathe, Jovan Komatovic, as well as my subsequent collaborators, Dr. Naama Ben-David, Dr. Igor Zablotchi, Dr. Javier Picorel, Dr. Pengfei Zuo, Dalia Papuc, Clément Burgelin, and Antoine Murat. Discussing and working with you on tough problems was definitely the most enjoyable part of my PhD. Special thanks to Antoine, not only for being my collaborator, but also for being my office mate. Finally, I express my gratitude to Dr. Pierre-Louis Roman who, despite not being a collaborator, was always available with insightful feedback and helpful suggestions.

During the PhD, the laboratory of Prof. Rachid Guerraoui has been such a cool place. I thank France Faille, our lab secretary, for her help in all sorts of situations and her positive attitude. Also, I thank our system administrator, Fabien Salvi, for his technical support. Finally, I thank all the folks in the lab, previous and present: Matej, George, Karolos, Arsany, Gauthier, Rafaël, Nirupam, Sadegh, John, Geovani, Nastia, Youssef, Martina, Manuel, Diana, and Matteo.

I would not have been able to complete this journey without the help of my friends: Xinrui, Greg, Mahyar, Adi, Khashayar, Andreas, Vasilis, George, Sahand, Diane, Ogi, Atri. You are an indispensable part of my life in Switzerland. Thanks a lot for adding so many good

memories and times of happiness to my life.

I am ultimately grateful to my family: my father, Anastasios, who taught me to always keep my composure during difficult and stressful situations, and my mother, Asimenia, who showed me that science is a lifelong endeavor. I cannot express enough gratitude to my brothers, Nicholaos and Dimitrios, for shaping me as an adult, and to my sister, Ioanna, for always listening to my worries.

Lausanne, July 10, 2023 A. X.

Preface

This thesis presents part of the research conducted as a doctoral assistant in EPFL's Distributed Computing Laboratory, under the supervision of Prof. Rachid Guerraoui, between 2019 and 2023. The main results in the thesis appear originally in the following publications (author names appear in alphabetical order).

- Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. uKharon: A Membership Service for Microsecond Applications. In *USENIX Annual Technical Conference (USENIX ATC)*, 2022.
- Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Dalia Papuc, Athanasios Xygkis, and Igor Zablotchi. Frugal Byzantine Computing. In *International Symposium on Distributed Computing (DISC)*, 2021.
- Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. uBFT: Microsecond-Scale BFT using Disaggregated Memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

Besides the above-mentioned publications, I was also involved in other research projects that resulted in the following publications:

- Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond Consensus for Microsecond Applications. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2020.
- Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online Payments by Merely Broadcasting Messages. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020.

Abstract

The landscape of computing is changing, thanks to the advent of modern networking equipment that allows machines to exchange information in as little as one microsecond. Such advancement has enabled microsecond-scale distributed computing, where entire distributed services (e.g., trading systems, key-value stores) execute within a few microseconds. As a result, these services manage to operate in a timescale that was not possible before, thus becoming the foundation for several other dependent services.

The high-level problem we study in this thesis is that of making microsecond-scale computing reliable, while simultaneously minimizing the latency overhead of achieving reliability. Reliability is essential, as it allows services to provide non-stop operation to their users in spite of failures. The recipe to achieving reliability is the well known technique of replication, yet replication can undesirably incur significant latency to microsecond-scale applications. We approach this problem from two fronts, as follows.

First, we observe a weakness in the way of achieving reliability in existing microsecond-scale applications. Such applications rely on replication but they only look at making replication efficient in the absence of failures. However, when failures occur, their latency cost increases by at least two orders of magnitude. This behavior results in latency spikes, making microsecond applications operate at milliseconds during periods of active failures. We set out to create uKharon, a membership service for the microsecond scale that aims at addressing these issues by exposing a simple and reusable interface that forms the basis for all microsecond-scale applications requiring high reliability. We additionally showcase how uKharon can be used to replicate a microsecond-scale application with minimal latency overhead.

Second, we take a step back and look at how to provide reliability under different failure models. uKharon assumes the standard failure model of crashes, yet in certain occasions failures can go beyond crashes. As such, we also look at Byzantine (i.e., arbitrary) failures, in an attempt to safeguard applications from spurious failures, such as data corruption and malicious behavior. However, dealing with Byzantine faults has always been associated with high cost, either due to the number of machines required for replication, or due to the necessary—yet computationally expensive—cryptographic signatures. We begin by studying the theoretical limitations of achieving frugality, i.e., using few replicas and minimizing signatures, in shared-memory distributed computing. With this knowledge, we then create uBFT,

Abstract

a Byzantine-resilient replication engine that relies on disaggregated memory, a technology enabled by modern networking adapters that realizes shared memory in practice. Due to its frugality, uBFT becomes the first system to offer Byzantine-resilient microsecond-scale replication.

Keywords: Byzantine-fault tolerance, crash-fault tolerance, consensus, consistent broadcast, failure detection, membership service, Remote Direct Memory Access (RDMA), non-equivocation, replication, state machine replication.

Résumé

L'informatique est en train de changer grâce au matériel réseau moderne qui permet aux machines d'échanger des informations en à peine une microseconde. Une telle avancée a permis à l'informatique distribuée de passer à l'échelle de la microseconde, où des services distribués entiers (par exemple, des systèmes de trading, des bases de données clé-valeur) s'exécutent en quelques microsecondes. En conséquence, ces services parviennent à fonctionner dans un délai qui n'était pas possible auparavant, devenant ainsi la base de plusieurs autres services dépendants.

Le problème global que nous étudions dans cette thèse est celui de rendre le calcul à l'échelle de la microseconde fiable, tout en minimisant simultanément le surcoût en latence engendré par la fiabilité. La fiabilité est essentielle, car elle permet aux services de fournir un fonctionnement continu à leurs utilisateurs malgré des pannes. La fiabilité est atteinte grâce à la technique bien connue de réplication, mais la réplication peut entraîner de manière indésirable une latence significative pour les applications à l'échelle de la microseconde. Nous abordons ce problème sous deux angles.

Tout d'abord, nous observons un problème dans la manière d'atteindre la fiabilité dans les applications existantes à l'échelle de la microseconde. De telles applications s'appuient sur la réplication, mais elles ne cherchent qu'à rendre la réplication efficace en l'absence de pannes. Cependant, lorsque des pannes surviennent, leur latence augmente d'au moins deux ordres de grandeur. Ce comportement entraîne des pics de latence, ce qui fait que les applications sensées fonctionner à l'échelle de la microseconde opèrent en réalité à l'ordre de la milliseconde pendant les périodes où des pannes subsistent. Nous avons créé uKharon, un service d'affiliation à l'échelle de la microseconde qui vise à résoudre ces problèmes en exposant une interface simple et réutilisable constituant la base de toutes les applications à l'échelle de la microseconde nécessitant une haute fiabilité. Nous montrons également comment uKharon peut être utilisé pour répliquer une application à l'échelle de la microseconde avec un coût minimal en latence.

Deuxièmement, nous examinons la fiabilité selon différents modèles de pannes. uKharon suppose le modèle standard de pannes franches, mais les pannes peuvent aussi aller au-delà des pannes franches. Nous examinons donc également les pannes Byzantines, dans le but de protéger les applications contre des comportements arbitraires telles que la corruption des

données et les comportements malveillants. Cependant, la gestion des pannes Byzantines a toujours été associée à un coût élevé, soit en raison du nombre de machines nécessaires à la réplication, soit en raison des signatures cryptographiques nécessaires qui impliquent des calculs coûteux. Nous commençons par étudier les limites théoriques de la réalisation de la frugalité, c'est-à-dire l'utilisation de peu de répliques et la minimisation de l'utilisation de signatures, dans l'informatique distribuée à mémoire partagée. Nous créons ensuite uBFT, un service de réplication résilient aux pannes Byzantines qui s'appuie sur une mémoire désagrégée, une technologie rendue possible grâce à du matériel réseau moderne qui réalise en pratique une mémoire partagée. Grâce à sa frugalité, uBFT est le premier système à offrir une réplication à l'échelle de la microseconde résistante aux pannes Byzantines.

Mots-clés : tolérance aux pannes Byzantines, tolérance aux pannes, consensus, diffusion cohérente, détection de panne, service d'affiliation, Remote Direct Access Memory (RDMA), non-équivoquation, réplication, réplication de machine d'état.

Contents

Introduction

Distributed systems form the backbone of computer systems in modern society. They are so pervasive, that we often tend to forget their existence and their importance in our lives. For example, today's online services, including social networks, search engines, video streaming, e-commerce and banking platforms are all instances of large scale distributed systems. Perhaps surprisingly, distributed systems also appear in many cyber-physical systems. The traffic lights at an intersection, the electronics in our cars, the wireless thermostats in our homes are only a few examples of distributed systems that go beyond the aforementioned large scale online services.

A distributed system is, more formally, a collection of several independent processing components that interact with each other via a networking substrate. The goal of such a system is to orchestrate its processing components in order to reach a collective goal, that is, to provide a certain service. Unfortunately, these comprising components are typically imperfect and therefore prone to failures, which can lead to inability of offering the desired service.

Building distributed systems is hard, especially due to the unpredictability and hostility of the environment where the system is operating in. Delays in the networking substrate delivering the messages, or in the processing components can lead to system failure, unless such behaviors are considered and addressed during the design phase of the system. Additionally, deliberate deviation of some processing components from their intended behavior can also lead to failure of the distributed system's offered service.

We address scenarios such as the ones mentioned above, by designing reliable distributed systems. Briefly, reliability refers to the system's ability in being available at the highest degree possible, in ensuring that the orchestration of processing components does not lead to data corruption or loss within the distributed system, and finally in tolerating failures of the distributed system's processing components.

Replication is the core technique that stands behind the reliability of distributed systems. In a replicated design, both the processing components executing the service, as well as the data comprising it, exist multiple times in replicated instances called *replicas*. The key insight is that by equipping the system with enough redundancy, it is possible to both replace a failing component with a redundant one, as well as use the redundant components to ensure that

deviation from the intended behavior is harmless, thus ultimately achieving reliability. Replication is typically achieved by means of a replication protocol, which ensures that replicas are kept synchronized regardless of failures, such as these of machines or networking equipment.

Replication, however is associated with additional overhead. More hardware is required in order to provide redundancy of the data and the processing components. At the same time, to build this redundancy, replication protocols involve additional processing steps, thus increasing the execution time of the service compared to a centralized service.

The impact of the increase in execution time due to replication depends on the time scale systems operate in. Modern hardware has enabled microsecond-scale computing, i.e., communication and processing time is now small enough that a centralized service can serve requests within a few microseconds. As a result, to design and implement reliable distributed systems that operate in microseconds requires rethinking replication. In such time-constrained environments, every microsecond counts, thus replication protocols need to be careful with how time is spent. For example, solutions that replicate slower than the microsecond scale are not appealing enough, as it is undesirable to provide reliability to a microsecond-scale (centralized) service without staying as close as possible to its pre-replication microsecond-scale characteristics.

Thesis Context

The data center presents a unique opportunity for innovation, as new hardware has led to a leap in data center performance. Today, typical network interconnects provide latency in the microseconds and throughput in the hundreds of Gbps, moving towards Tbps [140]. Simultaneously, computational power is greater than ever, thanks to the availability of CPUs with hundreds of cores, GPUs with thousands of cores, as well as FPGAs [2, 39, 149]. These advancements are making microsecond-scale computing a reality [5, 64, 90, 94, 148, 181].

However, a crucial consideration of microsecond-scale computing is its reliability. Several areas, such as finance (e.g., trading systems), embedded computing (e.g., control systems), and microservices (e.g., key-value stores) simultaneously strive for low-latency and reliability due to their critical nature. For such applications, failures should not jeopardize the availability and safety of the application. We approach reliability through replication. Our ultimate goal is to achieve microsecond-scale replication, in an attempt to lower the user-perceived cost of replication, considering two types of failures: simple (crash) failures, and arbitrary (Byzantine) failures.

Thesis Statement

In this thesis, we focus on replicating microsecond-scale distributed applications with the least amount of overhead. To accomplish this, we target—what we prove to be—the main

culprits of high latency in replication protocols: (1) reacting slowly to crash-stop failures, and (2) relying on many replicas or employing expensive cryptography when considering Byzantine failures. Towards this goal, we are inspired by the particular characteristics of a modern data center networking technology, namely Remote Direct Memory Access (RDMA), and show how these characteristics can be leveraged to our advantage. The end result is that we manage to improve over the state-of-the-art, while simultaneously keeping our solutions generic and not bound by the RDMA technology.

Thesis Roadmap and Contributions

In broad terms, this thesis spans across four parts, and makes three high-level contributions (pronounced through the use of vertical bars) as follows.

Part I—Preliminaries

In the first part of this thesis we cover essential concepts on distributed computing (Chapter 1) that are relevant in the subsequent parts, as well as provide a detailed overview of the subsequent parts (Chapter 2).

Part II—Microsecond-Scale Crash Fault-Tolerant Replication

In the second part of this thesis, we study replication for microsecond-scale applications under the crash-fault tolerance model.

> We present a generic and reusable membership service (Chapter 3), that (1) facilitates the replication of microsecond-scale applications, and (2) lowers their fail-over time.

Part III—Microsecond-Scale Byzantine Fault-Tolerant Replication

In the third part of this thesis, we turn our attention to Byzantine Fault-Tolerant (BFT) replication, which is capable of capturing faults that go beyond crashes.

> We identify and analyze the severity of cryptography bottlenecks of Byzantine consensus (Chapter 4) in the context of microsecond-scale applications, and provide theoretical lower bounds for this cost.

> We improve the common-case latency of Byzantine state machine replication in the data center (Chapter 5), thus bringing the cost of Byzantine-fault tolerance close to the cost of crash-fault tolerance.

Part IV—Epilogue

Concluding this thesis, we give a brief review of our contributions, discuss the limitations of our techniques and sketch avenues for future work (Chapter 6). Subsequently, we provide three appendices with supplementary details and proofs for the three respective contributions.

Part I Preliminaries

1 Distributed Systems Background

The evolution of computer systems over the last 75 years has seen rapid growth. Since the beginning of the modern computer era back in 1945, until the mid-80s, computers used to be bulky, expensive, and lacked connectivity.

However, after 1985, the landscape started to change in two directions. First, was the rapid development of more powerful microprocessors, reaching what we know today: multicore, 64-bit CPUs, offering high degree of parallelism, being relatively inexpensive, and requiring new software to exploit their power. Second, was the invention of high-speed computer networks. It enabled the deployment of local-area networks (LANs), which allow thousands of machines within a building or complex to be connected, as well as the deployment of wide-area networks (WANs), which enable the connectivity of hundreds of millions of machines around the globe.

The combination of these advancements made it possible to organise these interconnected computers into entire systems, called *distributed systems*. In fact, the possibility has now become commonplace practice, with data centers—accounting for 4% of the global electricity consumption as of today [101]—essentially hosting the world's storage and processing needs.

1.1 Definition

A *distributed system* is a collection of independent entities, called *processes* or nodes, that collaborate to solve a task, while appearing to its users as a single coherent system [171]. Typically, the processes collaborate by using *communication links* to exchange information with each other.

There are several reasons for studying distributed systems [104]:

- *Single-machine limit*: In many occasions, a single machine simply cannot store the data required for a computation, or the time it takes to execute the computation is impractical.

- *Reliability*: a distributed system is a perfect candidate to provide increased reliability, because it can replicate resources. Reliability has several aspects:
  1. availability, dictating that the resource is accessible at all times,
  2. integrity, dictating that the state of the resource is always correct, regardless of concurrent access to it by multiple processes,
  3. fault-tolerance, which is the system's ability to be impervious to certain failures and/or be able to recover from them.
- *Scalability*: is the capability of a system to grow and manage increased demand. A system may have to scale for various reasons, such as increased data volume or increased work (e.g., number of transactions). A scalable system should achieve this scaling while simultaneously addressing the problem of performance loss due to the increased cost of coordination among the growing number of machines.

1.2 Failure Models

A failure model [136] specifies how the components of a distributed system may fail. We separate failures in two categories, failures of processes and failures in the communication links.

Process failures. Several failure models have been studied in the literature [78, 112, 113, 145, 157], yet in this thesis we focus on two: crash-stop [112] and Byzantine [113] failures.

- *Crash-stop failure*: where a properly functioning process may fail by stopping to function from any instance henceforth. Importantly, when this happens the rest of the processes do not learn of this crash.
- *Byzantine failure*: where a process may exhibit any arbitrary behavior. Byzantine failures are much more severe than crash failures. Typically, distributed systems address them by relying on protocols that are more sophisticated than the ones used for crash-stop failures. This is because with Byzantine failures, a process cannot trust what individual processes are saying, as Byzantine processes are allowed to *equivocate*. Instead, a process can only trust a piece of information as long as it can confirm that enough processes have said or approved that information. Additionally, when considering this kind of failures, we typically rely on cryptography (e.g., unforgeable signatures [58], message authentication codes [95], cryptographic hashes) to restrict the type of claims a faulty processes can have. In this case, if a faulty process claims to have received a specific message from a correct process, then that claim can be verified.

Communication link failures. For the purposes of this thesis, we focus on two types of network failures, both of which may lead to network partitions [32, 72].

- *Benign failure*: which can manifest as either message reordering, duplication and loss.

- *Byzantine failure*: in which a communication link can exhibit any arbitrary behavior, including creating spurious messages and modifying the messages sent on it.

Depending on the failure model assumed, the algorithms used to solve any particular problem can vary dramatically. Thus, it is crucial to specify the failure model clearly. In the context of this thesis, Chapter 3 assumes crash-stop process failures and benign network failures. This is typical for distributed systems in the data center, since processes are operated by the same entity. Subsequently, Chapters 4 and 5 assume Byzantine failures for both processes and communication links, in order to push the reliability level of microsecond-scale distributed computing. The motivation behind the study of Byzantine failures is that distributed systems in the data center that aim at high reliability should also account for unplanned and unpredictable failures.

1.3 Synchrony vs. Asynchrony

In distributed systems, processes as well as communication links can be either synchronous or asynchronous.

In the synchronous case, processes execute in lock-step (i.e., they are synchronized) and the clock drift rate between any two processes is bounded. Typically, the synchrony is achieved with a form of distributed synchronization barrier. Thus, processes synchronize their execution by executing in *steps*, ensuring that no process begins the execution of the next step, until all processes have completed the execution of the previous step.

In the asynchronous case, there is no synchrony guarantee among processes and there is no bound on the drift rate of process clocks. Additionally, message delays on the communication links are finite but unbounded, and there is no upper bound on the time taken by a process to execute a step.

From a practical standpoint, systems tend to go both through periods of synchrony and asynchrony during their lifetime, a behavior that is captured by the *partial synchrony* model. Partial synchrony brings the best of both worlds, and famously makes consensus, a central part of this thesis, solvable [70]. Indeed, a system designed for asynchronous environments is inherently more robust than a system designed for synchrony, as it makes fewer assumptions on the permitted behavior of the processes and the network. Yet, synchronous systems usually exhibit better performance, as they enable process coordination with fewer exchanged messages compared to asynchronous systems.

1.4 Safety and Liveness

Distributed systems express the trade-off between synchrony and asynchrony using the notions of *safety* and *liveness* [107]. Informally, safety guarantees that nothing bad happens, i.e., it guarantees that our distributed protocol will never be broken. Liveness, on the other hand, guarantees that something good will eventually happen, i.e., it guarantees that our distributed protocol will not be stuck forever without making any progress.

In practice, as it is also the case throughout this thesis, we design distributed systems that are (1) safe under asynchony, in order to make sure that network or process delays do not lead to protocol violations, and (2) live under synchrony, in order to guarantee progress when the processes and the network are timely. Protocols that satisfy these two properties combine both robustness and performance.

1.5 Communication Models

So far, we have assumed processes in a distributed system communicate by exchanging messages over communication links. Formally, this means that processes use the *message-passing model* to communicate, thus they exchange information over a network, by *sending* and *receiving* messages.

Another means of communication, usually found in multiprocessor systems, is having processes communicate using *shared memory*. Essentially, with shared memory processes have access to a (common) shared address space, and communicate by *reading* and *writing* to shared variables, called *registers* [80, 106].

Conceptually, programmers find it easier to program using shared memory than message passing, since it gives the impression of a single monolithic memory, as in the traditional von Neumann architecture. This is why, in the context of distributed systems, we often emulate shared memory using message passing, despite the additional cost. In fact, it is well known [12] that communication via message-passing can be simulated by communication via shared memory and vice-versa, making the two models equivalent.

1.6 Remote Direct Memory Access

Remote Direct Memory Access (*RDMA*) [166] is a networking technology that fuses the message-passing and shared-memory models together and alleviates the cost of emulation. Applications communicate over RDMA by relying on primitives called *verbs*. There exist *one-sided verbs* that include READ, WRITE and Compare and Swap (CAS) verbs, and correspond to how RDMA exposes the shared-memory model abstraction. These verbs allow processes to access the memory of a remote machine without involving the CPU of the latter. Additionally, there exist *two-sided verbs*, such as SEND and RECV verbs, that correspond to how RDMA exposes

the message-passing model abstraction.

By implementing several layers of the networking stack in hardware and relying on kernel bypass, RDMA achieves microsecond inter-machine communication. It allows applications within the data center to communicate in as little as $0.9\mu s$ [92], and thus serves as first step towards microsecond-scale computing. This technology is supported by different fabrics such as Infiniband [166] and commodity Ethernet via RoCE [19].

1.7 State Machine Replication

A generic and popular approach to replication, which enables available and fault tolerant distributed systems, is *State Machine Replication* (SMR) [108]. With SMR, a service (e.g., a key-value store) is replicated across multiples machines (*replicas*), thus a failure of a replica does not prevent the service from being accessible due to the remaining alive replicas. The idea behind SMR is for the replicas to first agree on the order of incoming service requests, and then to execute the requests in that order and reply to the clients that invoked the service. Importantly, to reach agreement on the order of requests, the replicas in SMR employ a *consensus* algorithm.

The literature is rich in consensus algorithms, but conceptually all of them achieve the same goal: to decide on a single proposal (i.e., decide on which incoming client request to consider next) among multiple concurrent proposals. Consensus protocols are often leader-based [15, 37, 77, 84, 127, 134]: a replica designated as the *leader* orders the client requests, and forwards them to the other *follower* replicas to ensure agreement.

Though consensus and SMR are generally considered to be equivalent problems, in practice they differ in their memory consumption. Consensus must ensure agreement among participants indefinitely, so decided requests must be retained forever if a participant is unresponsive, thereby requiring unbounded memory. In contrast, SMR need not store all requests, as it cares only about replicating the application state, which can be finite even if there are infinitely many requests. Thus, SMR systems typically entail more complexity that consensus, as they need to adapt the underlying consensus algorithm to use finite memory [159].

Consensus and SMR are central to this thesis. Chapter 3 devises a consensus algorithm for the crash-stop model that leverages the shared-memory model exposed by RDMA, Chapter 4 focuses on Byzantine consensus and devises an algorithm that is economical in the number of replicas, and Chapter 5 extends this Byzantine consensus algorithm to a full-fledged practical SMR system.

1.8 Performance

When evaluating a distributed system, we are interested on certain metrics to assess its performance and quantify its efficiency. In this thesis we will focus on three metrics:

- *Latency*: the time required to transmit a message (e.g., a request) between two processes. It expresses how fast a distributed system can process messages.
- *Bandwidth*: the amount of data that can be transferred per unit of time across processes,
- *Throughput*: the number of message (e.g., requests) processed per unit of time. This metric expresses the system's ability to handle high loads.

2 Contributions Overview

The core content of this thesis is presented meticulously in Chapters 3 through 5. However, before delving into the details, we give a high-level overview below.

A membership service for the microsecond scale. We begin by studying replication for microsecond-scale applications under the crash fault tolerance model (Chapter 3). According to this model, processes in the system may stop executing, after which they become unresponsive. We approach replication through the lens of a membership service. A membership service is a generic and reusable distributed abstraction that aims at providing distributed applications with dynamic information regarding the processes that currently comprise the distributed system. With this piece of information, a replication protocol is significantly simplified, as it merely requires to focus on the distribution of data, rather on the complexities of data consistency upon failures.

We observe that a replication protocol suitable for the microsecond scale should possess two important properties. First, in the absence of failures, it has to incur minimal replication overhead. Failure-free execution is the common-case operational mode of data center applications, thus retaining to the maximum degree possible the microsecond-scale latency characteristics of the original (unreplicated) service is essential. Second, it is essential for a replicated service to exhibit stability even during uncommon situations, such as reconfigurations and failures. This is especially important as many microsecond-scale services running in the data center form the backbone of numerous cloud-powered services, such as analytics and trading systems, trying to meet ever-stringent tail latency requirements and maximum availability.

Our answer to systematically building replication protocols with the aforementioned properties is uKharon, the first ever microsecond-scale membership service that detects changes in the membership of applications and lets them failover in as little as 50µs. To achieve its goal, uKharon consists of (1) a multi-level failure detector, (2) a new consensus engine that relies on one-sided RDMA Compare&Swap, and (3) minimal-overhead membership leases, all tuned to

operate at the microsecond scale. We further showcase the power of uKharon by building a primary-backup replication protocol in the form of a replicated key-value cache.

The intrinsic limitations of microsecond-scale Byzantine-fault tolerance. We observe that traditional techniques for handling Byzantine failures are unsuitable for the microsecond scale: digital signatures are too costly, while the number of replicas used is uneconomical. Indeed, Byzantine fault tolerance (BFT) typically requires $3f+1$ replicas to tolerate up to f Byzantine process failures, which is considerably higher than the $2f+1$ replicas required tolerate f crash-stop process failures. As such, achieving microsecond-scale BFT replication becomes a challenging task which we approach methodically, initially from a theoretical perspective.

We begin by devising Byzantine-resilient algorithms that exhibit *frugality*, i.e., they reduce the number of replicas to $2f+1$ and minimize the number of signatures (Chapter 4). While the first goal can be achieved with relative ease, accomplishing the second goal simultaneously is challenging. We first address this challenge for the problem of broadcasting messages reliably, by focusing in two broadcast variants, Consistent Broadcast and Reliable Broadcast.

We then turn to the problem of consensus—the basis of replication—and argue that Consistent Broadcast is ideal for solving consensus with Byzantine failures. We present a consensus algorithm that works for $2f+1$ replicas and avoids signatures in the common case—properties that have not been simultaneously achieved previously.

Practical Byzantine fault-tolerant replication in microseconds. We leverage the aforementioned theoretical results to design uBFT, the first BFT SMR system to achieve microsecond-scale latency in data centers, while using only $2f+1$ replicas (Chapter 5). The Byzantine-fault tolerance provided by uBFT is essential in practice, as pure crashes are not the only type of failure that real-life systems exhibit.

uBFT is an example of the discrepancy between theoretical and systems work, overcoming the practical difficulties that are irrelevant to the theoretical analysis. To achieve $2f+1$ BFT, uBFT relies on a small non-tailored trusted computing base—disaggregated memory. At the same time, Consistent Broadcast turns out to be unimplementable in practice, as this abstraction requires replicas to use infinite memory in order to store and deliver all broadcast messages. uBFT overcomes this limitation by inventing a novel abstraction, called Consistent Tail Broadcast. Importantly, this abstraction is weaker than Consistent Broadcast, yet it is strong enough to implement SMR while bounding memory.

We implement uBFT using RDMA-based disaggregated memory and obtain an end-to-end latency of as little as 10 µs. This is at least 50× faster than existing state-of-the-art $2f+1$ BFT SMR. As a result, uBFT makes for a viable solution to critical microsecond-scale applications seeking efficient reliability. These application can now benefit from low latency, as well as

tolerance of arbitrary failures.

Supplementary material. We provide supplementary details and proofs of correctness for uKharon (Appendix A), uBFT (Appendix C), as well as for the theoretical work on which the latter is based upon (Appendix B) towards the end of the thesis.

Part II

Microsecond-Scale Crash Fault-Tolerant Replication

3 uKharon: A Membership Service for Microsecond Applications

In this chapter, we seek to minimize the replication latency for microsecond-scale applications under crash failures. To do so, we avoid solely designing a replication protocol, but we leverage replication to build a reliable membership service that can form the foundation for building replicated microsecond-scale applications. Membership services are ideal for everyone striving for genericity. They constitute a reusable distributed computing abstraction for the entire data center, powering microsecond-scale applications trying to meet ever-stringent tail latency requirements.

To this end, this chapter presents uKharon, a microsecond-scale membership service that detects changes in the membership of applications and lets them failover in a few tens of microseconds. uKharon consists of (1) a multi-level failure detector, (2) a consensus engine that relies on one-sided RDMA Compare&Swap, and (3) minimal-overhead membership leases, all tuned to operate at the microsecond scale. We showcase the power of uKharon by building uKharon-KV, a replicated Key-Value cache based on HERD [91]. uKharon-KV processes PUT requests as fast as the state-of-the-art and improves upon it by (1) removing the need for replicating GET requests and (2) bringing the end-to-end failover down to 53 µs, a 10× improvement.

3.1 Introduction

Despite substantial efforts in both hardware (e.g., InfiniBand/RDMA [87], RoCE [19], FPGA [40], Gen-Z [98], CXL [47]) and hardware-accelerated software [69, 86, 88, 90, 133, 175, 176, 180], building microsecond-scale applications with robust latency is very challenging [54].

Existing systems, such as key-value stores like Hermes [94], state machine replication [158] systems like Mu [5] and Hovercraft [99], and transactional systems like FaRM [64], process requests in a few microseconds in failure-free scenarios, but miss the microsecond envelope when handling failures. Mu and HoverCraft take 0.5ms and 10ms respectively to failover. Aguilera *et al.* [5] reported that Hermes has a failover of 150ms, while FaRM mentioned

ZooKeeper [84], a widely used distributed coordination service that offers at-best millisecond failover, for its membership management.

In other words, existing microsecond-scale applications fail to exhibit stringent *tail* latency, which is crucial in ensuring smooth and predictable operation. The tail refers to the latency of the slowest requests, and thus provides a limit to the maximum latency experienced by the consumer of such application.

We believe that a crucial step in making tail-tolerant microsecond applications is reacting fast to failures. We thus propose uKharon[1], a membership service tailored to the microsecond scale. Apart from acting as a distributed membership storage for (distributed) applications, uKharon monitors their nodes, detects their failures and changes their membership within 50 µs. When uKharon itself experiences a failure, it recovers within 64 µs. uKharon particularly benefits applications with efficient state transfer which can swap a faulty replica with a hot one in microseconds, for example via shadow replication. It targets cloud services that require seamless reconfiguration for fault tolerance and scalability, such as indexes, datastores and transactional systems.

The key to the performance of uKharon is the careful design of three fundamental components, all of which are inspired by Remote Direct Memory Access (RDMA) to operate at the microsecond scale. *First*, uKharon achieves microsecond failure detection by employing a multi-level failure detector. It distinguishes the failures related to the application (e.g., segmentation faults), from those related to the kernel (e.g., driver faults), and failures related to the hardware (e.g., RDMA NIC faults), employing for each a different failure detector. *Second*, uKharon decides on memberships using a consensus engine which solely relies on one-sided RDMA verbs. This engine takes advantage of RDMA Compare-and-Swap (CAS) to handle leader changes within 10 µs. *Third*, uKharon provides membership leases that add minimal overhead to the end application and last ~20 µs. As a result, our membership service combines typically opposing forces: having applications with low-overhead dynamicity in failure-free scenarios and very fast failover upon failures.

We showcase the benefits of our membership service by building uKharon-KV, a replicated in-memory KV-cache based on HERD [91]. It uses uKharon to track the set of nodes and react to node failures. We compare uKharon-KV against HERD+Mu [5] (i.e., HERD replicated by Mu), a system which—to the best of our knowledge—achieved the lowest replication latency to date. Our evaluation shows that uKharon-KV processes `PUT` requests as fast as HERD+Mu in failure-free periods. Moreover, thanks to its leasing mechanism, uKharon-KV manages to spare the replication of `GET` requests, an optimization that is algorithmically impossible in HERD+Mu. As a result, uKharon-KV `GET`s are 31.8% faster than HERD+Mu's. uKharon-KV, though, shines in the event of failures, achieving an end-to-end failover of 53 µs, improving on HERD+Mu's failover of 531 µs by up to a factor of 10.

[1] "u" stands for microsecond, and Kharon is the carrier of the souls of the dead in Greek mythology. It is pronounced ma · ka · ron.

In a nutshell, we present uKharon, the first ever membership service suitable for the needs of tail-tolerant microsecond applications. We make the following contributions:

- A multi-level failure detector for the microsecond scale.
- A consensus engine that relies on one-sided RDMA CAS to change leader within microseconds.
- Microsecond leases that have minimal impact on the performance of the end application.
- uKharon-KV, a replicated KV-cache which outperforms the previous state of the art.
- The source code of uKharon is available at https://github.com/LPD-EPFL/ukharon.

3.2 Background

3.2.1 Membership Service

To achieve resilience, long-lived distributed systems must be dynamic. Many systems [103, 105, 144, 161, 162] achieve dynamicity by relying on a coordination substrate, such as ZooKeeper [84] or etcd [146]. Among the various services (e.g., atomic locks, registers) these substrates offer, dynamicity is fundamentally addressed via their *membership service*.

A membership services offers dynamicity both in graceful executions and upon failures. In the former case, it serves join and leave requests issued by processes that want to become part of a distributed application or exit it. In the latter, it detects process failures and reacts to them. All these events are reflected through new configurations (called views or simply memberships). Essentially, a membership service acts as a storage of configuration information, keeping track of how the set of processes evolves, and exposes this information.

Typically, membership services rely on consensus [70] to establish a totally ordered sequence of views. Such services, including Zookeeper and etcd, offer strong semantics as all processes using the membership service transition through the same sequence of views.

Consensus-based membership services also offer real-time semantics. Apart from knowing the sequence of memberships, it is also important to know which is the (single) *active* membership. To understand why this real-time property is useful, consider the following example that incorrectly builds a cache storage solely relying on the sequence of memberships: The cache serves READ and WRITE requests. Initially, membership $M_1 = \{S_1\}$ designates server S_1 as responsible for the cache (i.e., S_1 stores it and serves requests). Eventually, a second membership $M_2 = \{S_2\}$ replaces S_1 with S_2. S_2, being part of M_2, proceeds with serving clients' requests and updates the content of the cache. At the same time, S_1 is unaware of M_2 and continues serving clients' requests as well. As a result, a client that is also unaware of M_2 and

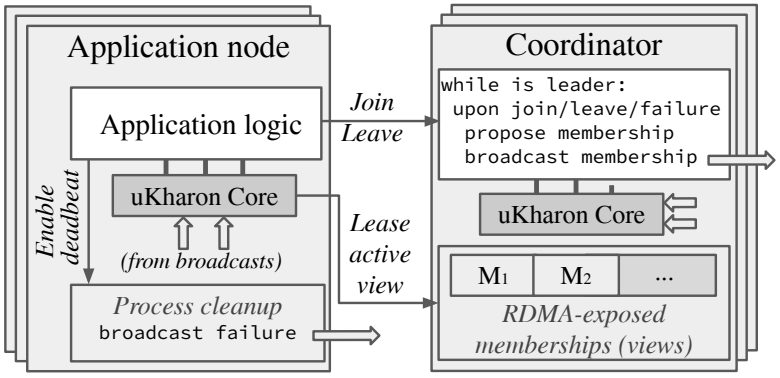

Figure 3.1: Overview of uKharon

reads from S_1 will get stale data. This example demonstrates a violation of consistency. It shows that total order of memberships does not provide any real-time guarantees by itself.

Membership services provide real-timeness by making outdated memberships nonoperational. A commonly used mechanism to achieve this property is the use of a distributed invalidation protocol. Another solution is to rely on leases. With leases, processes are forced to periodically check the active membership, execute operations in this membership, and abort operations that span over multiple memberships. uKharon provides real-timeness via leases.

3.2.2 Communication Model

uKharon is designed for data centers. It is safe under asynchrony and live under partial synchrony [66]. That is, to make progress, uKharon assumes a Global Stabilization Time (GST), unknown to the processes, such that from GST onwards there is a bound Δ on communication and processing delays. This is is a realistic assumption, as data center fabrics are not asynchronous in practice [6, 119, 177]. Additionally, our system relies on bounded clock drift for safety, i.e., durations are approximately the same across all processes. uKharon also assumes *crash-stop* failures. Finally, we assume that network partitions, which affect uKharon's liveness, are eventually resolved by the data center administrator.

3.3 Design Overview

3.3.1 Architecture

Figure 3.1 gives an overview of uKharon. Our system, as a membership service, runs on application nodes as well as a set of dedicated nodes called *coordinators*.

Central to uKharon is uKharon Core, a single-threaded library that hosts monitoring functionalities of the membership service. This includes detecting failures of member nodes (including coordinators), listening for failures and new memberships, as well as renewing leases. The

application receives these events via thread-safe accessors: a stream of failures, a stream of memberships and a method `Active(M)` → *bool* which checks whether a given membership `M` is active.

The generation and storage of memberships is delegated to coordinators. Coordinators achieve fault tolerance through consensus. One of them is the leader, which processes join/leave requests from both application nodes and coordinators, proposes new memberships and broadcasts decided memberships which are picked up by the uKharon Core instance running on every node. The rest of coordinators help the leader decide and replicate the sequence of memberships. Finally, coordinators assign each member a unique identifier.

Running uKharon Core on both application nodes and coordinators helps bootstrap the membership service. uKharon Core learns about the new memberships from coordinators, but coordinators require the membership service to learn about each other. Similarly, coordinators rely on uKharon Core to detect failures of application nodes or themselves.

Part of uKharon's failure detection logic resides in the kernel, outside of uKharon Core. It consists of a kernel module hooked to Linux's process cleanup routine. This module can be enabled by the application logic and broadcasts a failure notification (called *deadbeat*) when the application crashes.

New memberships are merely broadcast by coordinators, putting the burden of detecting the active membership to the application nodes. uKharon Core is responsible for bringing real-timeness to applications. It reads the RDMA-exposed memberships at a majority of coordinators to determine whether a membership has been superseded by a new one or whether it is still active. The active membership is leased for a limited amount of time, in our case ~20 µs.

3.3.2 Communication

uKharon relies extensively on the performance of today's RDMA-enabled fabrics to achieve its microsecond latency target. It leverages one-sided RDMA verbs, two-sided ones (i.e., HERD-style RPC [91]), as well as RDMA Multicast. Coordinators run consensus using RDMA Reliable Connections (RCs). In particular, coordinators establish all-to-all connections among themselves and communicate using RDMA READ, WRITE and CAS. Additionally, coordinators use RDMA Multicast, which is backed by RDMA Unreliable Datagrams (UDs), to notify all nodes about new memberships. uKharon also uses RDMA Multicast to emit failure notifications. uKharon Core relies on RDMA READs over RCs to retrieve the active membership from coordinators and to detect the failure of remote nodes. Finally, processes send *join* and *leave* requests to the coordinator leader using RPC.

3.3.3 Challenges

Our system is designed for applications that operate and failover at the microsecond scale. To do so, uKharon meets two important design goals. First, it itself operates at the microsecond scale, meaning that it is able of changing the active membership within as few as 50 µs. Second, we ensure that uKharon Core has minimal performance overhead on the end application it is bundled with. To meet these goals, uKharon is structured around three major components:

Failure detection. Efficient failure detection is the first step towards fast failover. Conventional wisdom suggests that there is a trade-off between the speed and accuracy of a failure detector. We work around this limitation by building a hierarchy of RDMA-tailored failure detectors suited for the microsecond scale. Our hierarchy detects failures within a few tens of microseconds, as we explain in Section 3.4.

Consensus engine. The second step of failover is agreeing on the new membership. Existing leader-based consensus engines, although optimized for the microsecond scale, struggle to change their leader at this time scale. In Section 3.5, we explain how our microsecond consensus engine changes leader in microseconds. This gives our design the unique property that a coordinator failure—especially failure of the coordinator leader—has negligible effect on the failover time.

Leases. As far as the membership service is concerned, the last step towards failover is updating the active membership. However, the new membership cannot become active before leases on previous memberships have expired. Thus, the longer the leases, the higher the failover time. On the other hand, short leases can result in application overhead, as they have to be checked in the application's critical path and renewed in time before expiring. In section 3.6, we explain how uKharon manages to have ~20 µs leases with virtually no cost for the end application and how leases can scale to hundreds of machines for an extra ~20 µs.

3.4 Microsecond Failure Detection

uKharon relies on microsecond failure detection to notify nodes about member failures and to trigger the generation of new memberships. In this section, we describe uKharon's failure detection scheme.

3.4.1 Multi-Level Failure Detection

A practical failure detector aims at being as complete and as accurate as possible. A complete and accurate failure detector is able to detect all failures and not have false positives, respectively. Completeness without accuracy causes problems in practice, as false positives

trigger new memberships which require distributed applications to take further action (e.g., rebalancing data among nodes).

Commonly, failure detectors rely on timeouts for their operation. However, timeouts are hard to set correctly: if they are too low, the failure detector may experience instability (e.g., oscillating behaviors). That explains why most systems set the timeouts to a safe high-enough value. In the microsecond scale this problem is magnified, as small execution delays (e.g., kernel jitter) can take several microseconds.

Our failure detector follows a pragmatic approach: it avoids timeouts when possible. To achieve this, we are inspired by Falcon [119], and identify four levels of failures: (1) *userspace failures* (e.g., segmentation faults, out of memory errors, uncaught exceptions) that cause the application to abort, (2) *kernel failures* (e.g., cores hanging in the kernel, kernel oops caused by driver crashes) that impede the application's execution, (3) *catastrophic failures* (e.g., power failures, RDMA NIC failures) that prevent communication with the application's host, and (4) *byzantine failures* (e.g., stack overflows, mercurial cores [82]) that affect the application state. Each of the first three levels is handled by uKharon via a specialized failure detector. uKharon does not address Byzantine failures, which is the topic of Chapters 4 and 5.

3.4.2 uKharon's Failure Detectors

We now explain how uKharon's specialized failure detectors work, depending on the type of failure.

Userspace failures. They are handled by the Linux kernel. The application registers to the kernel to enable a *deadbeat*, which is a failure notification broadcast by the kernel upon the death of the process. This registration happens by means of the `prctl` system call that the application calls early in its execution. The system call includes the node's identifier and modifies the process descriptor (Linux's `task_struct`) with a flag that the kernel checks during the *cleaning routine* of the process. In Linux, when a process crashes, control is transferred to the kernel which starts executing the process cleaning routine. If the flag is set, the kernel broadcasts a failure notification that includes the specified identifier. To achieve this functionality, we extend the `prctl` system call and modify the process cleaning routine that is part of the kernel's `exit` system call. The task of broadcasting the crash notification is delegated to a kernel module. This module uses the kernelspace RDMA driver to broadcast crash notifications which are polled by all instances of uKharon Core. As this failure detector does not use timeouts, it has no false positives.

Kernel failures. To detect application failures caused by the kernel, we rely on the way RDMA is handled in userspace. An application registers memory to an RDMA device by issuing `ioctl` system calls on a file descriptor. By design, the Linux kernel destroys that file descriptor and

thus disables remote access to this memory at the end of the process' cleaning routine. If this cleaning routine runs, the failure is caught by the previous failure detector. Otherwise, the memory will remain remotely accessible while the execution of the application is suspended (and the kernel is dying).

For the operation of this failure detector, processes are arranged in a logical ring where every process monitors its successor. Our system uses a local heartbeat counter in a similar fashion to Mu's detector [5]. uKharon Core increments this counter to indicate that the process is alive. This counter is read by the predecessor process. If a process RDMA-reads the same value twice, it reports its successor as having failed.

A process would be wrongly detected if it were unable to increment its counter between two consecutive reads. Thus, we take special care to ensure that processes always increment their counters faster than the time delay between two consecutive reads. Importantly, we deploy (the single-threaded) uKharon Core in its own dedicated physical core. We resort to a custom kernel compiled with the `NO_HZ_FULL` option, which disables regular timer interrupts [57] on the dedicated core and thus reduces the kernel jitter towards uKharon Core. Additionally, we boot this kernel with the `isolcpus` parameter, which prevents other userspace processes from sharing the dedicated core with uKharon Core. In experiments, the interval we observed between two counter increments under heavy load was 5 µs most of the time and never more than 15 µs. To account for unexpected jitter (e.g., thermal throttling), we make processes wait 30 µs after the completion of an RDMA READ before issuing the next one. As RDMA READs are issued sequentially, network delays do not negatively impact the accuracy of this failure detector.

Catastrophic failures. uKharon relies on a timeout-based scheme to detect failures that prevent machines from communicating. We set the timeout to 1ms, which is $2-3$ orders of magnitude higher than the common case latency of modern data center fabrics. As reported by Li *et al.* [121], 1ms is safe even in case of network congestion.

The detector works by having processes periodically broadcast a heartbeat and poll for heartbeats from others. Processes keeps track of the set of processes they recently received a heartbeat from. They compare this set with the current membership and report which processes they consider failed to the coordinator leader. Then, the leader constructs a connectivity graph based on the reported link states and changes the membership to approximately match the maximum clique in which it is included. Thus, our membership service enforces all-to-all connectivity among the members and does not expose any information regarding network partitions. A systematic treatment of network partitions is out of our scope.

The first two detectors broadcast failure notifications over RDMA-multicast, which offers better scalability than broadcasting using Reliable Connections. Nevertheless, RDMA-multicast is backed by Unreliable Datagrams, thus failure notifications can be lost under high network load. Dropping these notifications is safe, as uKharon-Core rebroadcasts a failure notification

until a new membership excludes the failed node.

3.5 Microsecond Consensus

In this section, we present a state-of-the-art consensus engine that is tailored for the needs of uKharon and powers its coordinators. Our engine is efficient regardless of failures: in the absence of failures, it decides in one RDMA delay (by issuing an operation to a majority of processes in parallel), while it decides in one additional RMDA delay in the event of a failure. It uses a slightly modified version of Paxos based on the observation that the original algorithm contains RPCs that can be emulated with RDMA CAS operations. In the rest of the section, we intuitively describe our consensus algorithm and discuss implementation details. Appendix A.1 provides its pseudocode and a proof of its correctness.

3.5.1 Consensus and Paxos

Consensus is a fundamental problem in distributed computing. Informally, each process proposes a value and eventually all processes irrevocably agree on one of the proposed values. Processes agree on a sequence of values and totally order them by running multiple instances of consensus.

Several algorithms solve consensus in the partially synchronous model. Many are variants of Paxos [110]. In Paxos, processes are divided in two groups: *proposers* and *acceptors*. Proposers *propose* a value for decision and acceptors *accept* some proposed values. Once a value has been accepted by a majority of acceptors, it is decided by its proposer.

Intuitively, Paxos is split in two phases: the *Prepare* phase and the *Accept* phase. During these phases, messages from the proposer are identified by a unique *proposal number*. The Prepare phase serves two purposes. First, the proposer gets a promise from a majority of acceptors that another proposer with a lower proposal number will fail to decide. Second, the proposer updates its proposed value using the accepted values stored in the acceptors. This way, if a value has been decided, the proposer will adopt it. The prepare phase can also *abort* if any acceptor in the majority previously made a promise to a higher proposal number. If the proposer manages to complete the Prepare phase without aborting, it proceeds to the Accept phase. In this phase, the proposer tries to store its value in a majority of acceptors. If it succeeds (i.e., a majority accepted the value), it decides on that value.

3.5.2 One-Sided Paxos

Paxos uses RPC in a very specific form. The acceptors' state consists of only three variables: `min_proposal`, `accepted_proposal` and `accepted_value`. In both phases, acceptors atomically update these values based on the proposer's input and return some of them.

```
1 # Paxos's RPCs pattern
2 def rpc(x):
3   if compare(x, state):
4     state = f(state, x)
5   return proj(state)
```

```
1 def cas-rpc(x):
2   expected = fetch_state()
3   if not compare(x, expected):
4     return proj(expected)
5   move_to = f(expected, x)
6   old = state.cas(expected, move_to)
7   if old == expected:
8     return proj(move_to)
9   abort
```

Algorithm 3.1: Paxos's RPCs turned into CAS-based RPCs.

Algorithm 3.1 proposes an obstruction-free transformation to turn Paxos's RPCs into purely one-sided conditional writes using RDMA CAS. Paxos's RPCs follow the pattern seen in `rpc`. The acceptor executing the RPC compares the received value `x` to its state (stored in `state`). If the comparison is successful, the acceptor updates its state (shown with function `f`) using the provided value `x`. Finally, the acceptor unconditionally returns part of its state (shown with function `proj`).

The pattern presented in `cas-rpc` allows RDMA to emulate `rpc` while solely relying on one-sided verbs. Opposite to `rpc`, which is executed on the acceptor's side, `cas-rpc` is executed on the proposer's side. To execute the one-sided RPC, the proposer first needs to know the `state` that is stored in the memory of the acceptor. This value can either be guessed (e.g., using a previous value of `state`) or fetched (e.g., using RDMA READ, as shown in line 2). Then, the proposer executes the comparison locally (line 3) and decides whether to continue or terminate. If the comparison succeeds, the proposer proceeds with updating the state of the acceptor. It is this update that utilizes CAS [2]. In line 7, if the CAS succeeds, the acceptor's `state` has been updated successfully with the value of `move_to`. Otherwise, `state` remains unchanged.

When the RDMA CAS succeeds, i.e., in the absence of contention, both `rpc` and `cas-rpc` are equivalent (see Appendix A.1.2). However, if the RDMA CAS fails, `cas-rpc` will abort while `rpc` would not. In this case, `rpc` and `cas-rpc` are not equivalent, but this does not violate the correctness of Paxos. The reason is that Paxos tolerates an arbitrary number of proposer failures and that aborting the RPC and starting over is indistinguishable from such failure.

3.5.3 uKharon's Consensus Engine

We now explain how to make the variant of Paxos described in Section 3.5.2 practical and compare it with Mu [5], a state-of-the-art consensus engine.

[2] As a reminder, `variable.cas(expected, new)` atomically checks if `variable` equals `expected` and sets `variable` to `new` if this is the case. The operation always returns the initial value of `variable`.

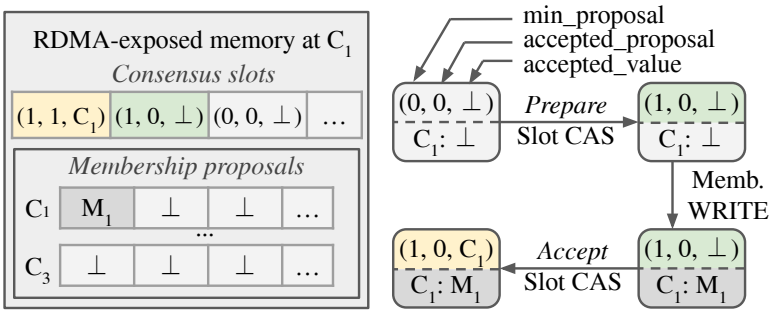

Figure 3.2: uKharon's Consensus Engine with its RDMA-exposed memory for multiple instances of consensus (left) and a state machine for a single instance of consensus (right).

Practical Considerations

Leader election. To avoid the contention rising from multiple concurrent proposers, our consensus engine adopts the same leader election scheme as Mu. The process with the lowest identifier among the coordinators considered alive is elected as the leader. In the event of a partial network partition, this scheme can elect multiple leaders. For example, if coordinator C_2 is the only one unable to reach C_1, it will think of itself as the leader, while other coordinators will consider C_1 as their leader. Having multiple leaders cannot lead to multiple values being decided, i.e. safety is always preserved. Leader contention can, however, prevent the engine from being live. Thus, a leaders that fails to decide uses a randomized backoff before proposing until the partition is resolved.

Pre-preparation. Coordinators decide on a sequence of values by running consensus on a sequence of *slots*, as shown in Figure 3.2. It requires two RDMA delays for each slot: one for the Prepare and another for the Accept phase (shown with horizontal arrows in the figure). A stable leader can prepare slots in advance and only run the Accept phase to decide. In this case, the leader decides in a single RDMA delay. The leader uses the time spent waiting for the Accept phase to complete on a slot to run the Prepare phase for the next one. Thus, it always maintains one pre-prepared slot (depicted in the second consensus slot of Figure 3.2), with no latency overhead. Switching to the new leader requires re-preparing the next slot. As an optimization, the new leader predicts that the last slot had been prepared by the previous leader and uses this prediction as the expected value of the RDMA CAS. With this approach, the new leader manages to re-prepare the next slot in a single RDMA delay instead of two.

CAS size limitation. Algorithm 3.1 assumes that the consensus state fits within a single CAS. Current RDMA NICs only support CAS up to 8 bytes. We set both `min_proposal` and `accepted_proposal` to be 2 bytes each. The remaining 4 bytes are dedicated to the `accepted_value`.

Proposal fields will overflow after 2^{16} attempts. In such an unlikely scenario, our consensus

engine falls back to traditional RPC: Once the RDMA-exposed `min_proposal` of an acceptor reaches $2^{16} - |\Pi|$, proposers switch to RPC to communicate with this specific acceptor. Acceptors check `state` and, if it is above the threshold, initiate the standard RPC version of Paxos with the `min_proposal`, `accepted_proposal` and `accepted_value` variables initialized to match `state`.

Our consensus engine uses indirection to overcome the limited size of the `accepted_value` and store uKharon's memberships. Instead of deciding on the membership itself, coordinators decide on its location in memory. First, the proposer RDMA-writes the membership to a part of acceptors' memory dedicated to membership proposals (see Figure 3.2) to which it has exclusive write access. Then, the proposer runs the Accept phase where it proposes its own identifier (C_1 in the figure). If the Accept phase succeeds at a majority of acceptors, then the proposer decides. Thanks to the FIFO semantics of RDMA RCs, if the last RDMA operation (i.e., the Accept phase CAS) succeeds, the previous RDMA operation (i.e., storing the membership with an RDMA WRITE) also succeeded. The two RDMA operations combined do not execute atomically, yet a coordinator cannot have accepted an identifier without knowing its associated membership.

Comparison with the State-of-the-Art

Many systems, such as Mu[5], DARE [147] and APUS [173] study consensus over RDMA. They primarily focus on improving the throughput and latency of common case executions, thus achieving consensus in a few microseconds. However, these systems have failovers ranging from 0.5 ms (in Mu) to 10s or 100s of ms (in DARE and APUS, respectively).

Mu has the best performance in failure-free executions among competition as it solves consensus in ~1.4 µs. It relies extensively on RDMA permissions. During its Prepare phase, a proposer asks acceptors for the exclusive write permission to their memory and waits for a majority of replies. This step guarantees that only one proposer can write to an acceptor at a time. In the Accept phase, the proposer decides by merely writing to a majority of acceptors. As acceptors give write permissions to a single proposer at a time, no two concurrent proposers can successfully write to a majority of acceptors and decide on different values. Since WRITE is the most efficient RDMA verb and the Prepare phase runs only once per leader change, Mu is optimal in failure-free executions.

The Accept phase of our algorithm relies on a WRITE followed by a CAS. Importantly, these one-sided operations have lower tail latency compared with the two-sided verbs present in DARE and APUS. The CAS increases the decision time from 1.4 µs to 2.9 µs compared with Mu. When it comes to a leader change, Mu's permission change mechanism requires approximately 250 µs, since it constitutes a control path operation that involves a system call and a reconfiguration of the NIC. In our consensus engine, the additional CAS lets coordinators change leader in under 10 µs. Thus, our algorithm is designed for short tail latency and makes the failure of the coordinators' leader no more important (latency-wise) than the failure of any

other node.

3.6 Microsecond Real-Timeness

In addition to reacting to failures and deciding on views, uKharon lets applications track the active membership via the `Active` method. While this information is essential for consistency, it must not burden the end application. In this section, we describe the challenge of making `Active`'s overhead negligible while preserving microsecond view changes.

3.6.1 The Active Method

uKharon exposes real-timeness to end applications via the `Active(Membership)→bool` method. If `Active(M)` returns `true`, we say that M is *active* at some point between the call and return of the method. `Active` satisfies three important properties. First, there are no two overlapping active memberships. Second, after a membership M is active, no memberships older than M become active. Third, the active membership converges to the latest decided membership.

Intuitively, processes use the `Active` method to determine the membership they should be executing operations in. When coordinators decide on a new membership M', a process p may stay in an older membership M due to a delay in receiving M'. Calling `Active(M)` will eventually return `false` at p, thus letting it realize that it misses the latest membership M'. To ensure consistency, an application typically calls `Active` once before starting an operation and a second time before committing it, only committing if both calls return `true`.

3.6.2 Leases

uKharon uses leases for efficiency. We proceed incrementally, first describing an implementation of `Active` without leases, before moving to a more efficient lease-powered scheme.

The basic implementation of `Active` requires communication in every invocation. Let M be the k-th membership decided by the coordinators and assume a process p invokes `Active(M)`. In essence, `Active` declares that M is active if it can conclude that no newer membership M' has been decided. To this end, the process RDMA-reads the $k+1$-th consensus slots at coordinators and waits for a majority of replies. If all replies are empty, then the $k+1$-th membership has not been decided, meaning that M is (still) active at some point between the invocation and return of the method. If, on the other hand, at least one of the replies is non-empty it is inconclusive whether M has been superseded by M'. In case M' has been decided before p issues the READs, then at least one of the replies must be non-empty, but the opposite is not always true. For safety, `Active` returns `false` if at least one of the READs on the next consensus slot is non-empty.

```
1  leased_membership = ⊥; t_start = 0; t_end = 0

3  def Active(M) → bool: # M is always a decided membership
4    t = hw_timestamp()
5    if leased_membership != M:  # First-time lease on M
6      if majority_active(M):
7        leased_membership = M; t_start = t + δ; t_end = t_start
8    else: # Check/extend lease on M
9      if t in [t_start, t_end): return True
10     if majority_active(M):
11       t_end = t + δ
12       return t > t_start
13   return False
```

Algorithm 3.2: Leased active membership.

A lease refers to a membership and has a start and an expiration date. A lease guarantees its holder that its associated membership will remain active until it expires. In our system, leases are created by uKharon Core and last $\delta \approx 20$ µs.

Algorithm 3.2 provides an efficient alternative implementation of `Active` that relies on leases to reduce communication. It starts by taking a hardware timestamp t (line 4) and then checks if a lease on M already exists (line 5). If no lease exists (lines 6-7), the method checks for a newly decided membership by contacting a majority of coordinators. If no membership newer than M could have been decided (i.e., all replies are empty), it creates a lease on M (line 7) that starts at $t+\delta$ and has no duration. This prevents overlapping active memberships since any lease that processes could hold on a previous membership $M' < M$ will have expired before M becomes active. In case a lease on M already exists, the method tries to use it in order to avoid reaching the coordinators (line 9). If it cannot use it, it tries to extend the lease (line 11) by checking the coordinators. It returns `True` only if leases on previous memberships have expired (line 12), which takes—in the worst case—δ to happen. As a result, leases affect the speed at which memberships can change, justifying the desire for a small lease duration. Section 3.7 demonstrates that leases of $\delta \approx 20$ µs are feasible in practice.

This efficient implementation of `Active` renews its lease on demand. As long as its lease is valid, the method merely takes a hardware timestamp—which takes a few tens of nanoseconds—and returns immediately without reaching the coordinators. The latency overhead of `Active` to the application that invokes it is thus very low. Communication with the coordinators is only necessary when leases expire and have to be renewed, which results in a spike in `Active`'s latency. In practice, uKharon Core renews leases in the background to ensure that—when the membership remains unchanged—`Active` is not delayed by the calls to `majority_active`.

uKharon does not rely on operational leases for either liveness or safety. Timely renewal of leases is only a way to reduce the latency of `Active` as Algorithm 3.2 would work even with zero-duration leases. uKharon relies on bounded clock drifts for safety, as opposed to clock synchronization. This ensures that durations are approximately the same across all processes, thus preventing overlapping memberships. Appendix A.3 includes a microbenchmark evaluat-

ing the clock drift of actual hardware and gives an overestimated drift that is no more than 0.001% of the lease duration. Thus, clock drift is accounted for by making leases last a few nanoseconds less than their nominal value. As drift is reset on each lease renewal, it does not accumulate over time. Therefore, no matter how long a system is up for, its operation remains unaffected by the clock drift. A proof of correctness of uKharon's leases is given in Appendix A.2.

3.6.3 Extensions

Adaptive leases. So far, we have assumed a fixed lease duration δ. Network delays greater than δ render leases useless as, every time the lease is extended (line 11), t_{end} is always in the past. In this case, `Active` always contacts the coordinators. In order to work under partial synchrony and avoid this scenario, we extend the leasing mechanism as follows: Coordinators store the lease duration for a given membership along with the membership itself. An application node that wants to increase the lease duration contacts the coordinator leader. This results in a new *compatible* membership that is identical to the previous one apart from the lease duration. Compatible memberships receive special handling by uKharon Core in order to ensure that—when going from one compatible membership to another—`Active` does not wait for leases on the previous membership to expire. Also, if the latest membership M is not compatible with the previous one, invocations to `Active(`M`)` return false until all possibly ongoing leases on previous memberships have expired.

Lease caches. `Active` reaches a majority of coordinators to renew its lease, which scales badly as the number of application nodes increases. uKharon solves this issue with an intermediate lease renewal layer, the *lease caches*. These caches use the `Active` method to lease memberships for Δ (by reading from a majority of coordinators). In turn, application nodes use leases that last for δ and a modified version of `Active`. This version differs from the one presented in Algorithm 3.2 in the `majority_check` calls, which are replaced with RPCs to a *single* lease cache. As a result, application nodes reduce the communication cost required to renew their lease by a factor of—at least—3 (the typical number of coordinators). However, lease caches increase the failover time of applications by at least Δ. The reason is that when the coordinators change the membership, the `Active` method of caches waits Δ before making the new membership active. At the same time, the `Active` method of application nodes that is directed to some lease cache, waits δ before making the new membership active. Thus, the overall time from the moment a new membership is decided until application nodes start using it jumps from (at least) δ to (at least) $\Delta + \delta$.

3.7 Evaluation

We evaluate the various performance traits of uKharon and verify its suitability as a membership service for microsecond applications. We aim to answer the following:

- How much does uKharon increase the latency of end applications and what is its impact on their throughput?
- How fast does uKharon respond to failures?
- How can uKharon be leveraged to build replication protocols and what performance can they achieve?

CPU	2x Intel Xeon Gold 6244 CPU @ 3.60 GHz (8 cores/16 threads per socket)
NIC	Mellanox ConnectX-6 MT28908
Switch	Mellanox MSB7700 EDR 100 Gbps
OS/Kernel	Ubuntu 20.04.2 / `5.4.0-74-custom`
RDMA Driver	Mellanox OFED `5.3-1.0.0.1`

Table 3.1: Hardware details of machines.

We evaluate uKharon in a 8-node cluster, the details of which are given in Table 3.1. The custom kernel sets the `NO_HZ_FULL` option and uses the `isocpus` boot parameter, as explained in Section 3.4.2. The dual-socket machines have an RDMA NIC attached to the first socket. Our experiments execute on cores of the first socket using local NUMA memory.

Our implementation measures time durations using the `clock_gettime` function with the `CLOCK_MONOTONIC` parameter. The function uses the TSC clocksource of the Linux kernel, which offers efficient and accurate timestamping [151]. Appendix A.3 discusses details regarding the drift and synchrony of TSC in symmetric multiprocessing (SMP) systems.

Finally, in all experiments we deploy 3 coordinators.

Applications. We integrate uKharon with HERD [91]. HERD is a non-replicated microsecond-scale RDMA-based KV-cache. Clients send requests to a HERD server by RDMA-writing to a dedicated buffer that the server has allocated for them. Requests contain an 8-byte key and are either `PUT`s or `GET`s. `PUT`s additionally contain the value to be stored for the specified key. The server discovers new client requests by polling its local memory, executes the requests locally and then replies to the clients using RDMA UDs. We also leverage uKharon to build uKharon-KV, an extended version of HERD which supports replication. We compare our solution with HERD replicated by Mu (HERD+Mu) [5] which—as far as we know—offers the lowest replication latency to date.

Implementation effort. We developed our own RDMA framework to implement uKharon. uKharon Core and the consensus engine span 4 448 and 1 324 lines of C++, respectively. The kernel module of the deadbeat failure detector is 404 lines of C. uKharon-KV extends HERD by 1 498 lines of C++. The only unimplemented features are clique-based memberships (Section 3.4.2) and adaptive leases (Section 3.6.3).

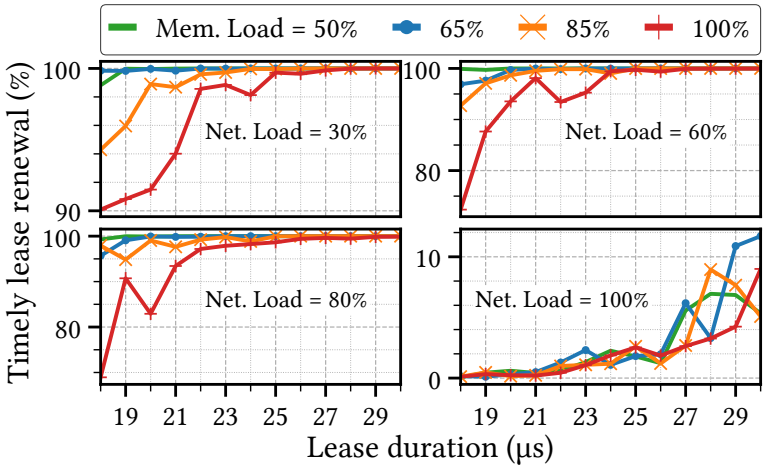

Figure 3.3: Percentage of timely lease renewal depending on the lease duration, network load and memory load.

3.7.1 Overhead Induced by uKharon

Latency overhead. Applications bundled with uKharon Core rely heavily on its `Active` method. As long as (the background running) uKharon Core renews the lease on the active membership in time, the `Active` method adds negligible latency overhead to the application. We experimentally determine that the 99th percentile latency for invoking `Active` is 38 ns when the lease is renewed in time, which is the time it takes to fetch the hardware timestamp and compare it with the expiration date of the lease. Fluctuations in the network's latency or execution delays when uKharon Core renews the lease (e.g., due to cache misses) induces additional latency to the application, as explained in Section 3.6.2.

Figure 3.3 shows how the duration of leases affects their timely renewal. We run 1-minute experiments under a steady membership with 32 lease renewers contacting coordinators directly and lease durations ranging from 18 to 30 µs. Each machine has a maximum memory bandwidth of 480 Gbps and a maximum network bandwidth of 100 Gbps. We apply variable network and memory load by running `stress-ng` [97] and `perftest` [16] on the first socket of our machines.

When the network load is maximum (bottom right figure), less than 12% of the calls to `Active` return immediately, irrespective of the memory load. For network loads of 30–80% (other figures), the memory load progressively affects lease renewal. Maximum memory load causes expired leases when lease duration is shorter than 27 µs. For most other configurations, a duration greater than 23 µs suffices. For example, with 80% network and 50% memory load, lease renewal fails 0.0011% of the time, which corresponds to `Active` inducing latency every 300 out of 1.5 billion invocations. In other words, the 99.999th percentile of `Active`'s latency is 2 µs.

We get similar (omitted) results when an application renews its leases through lease caches.

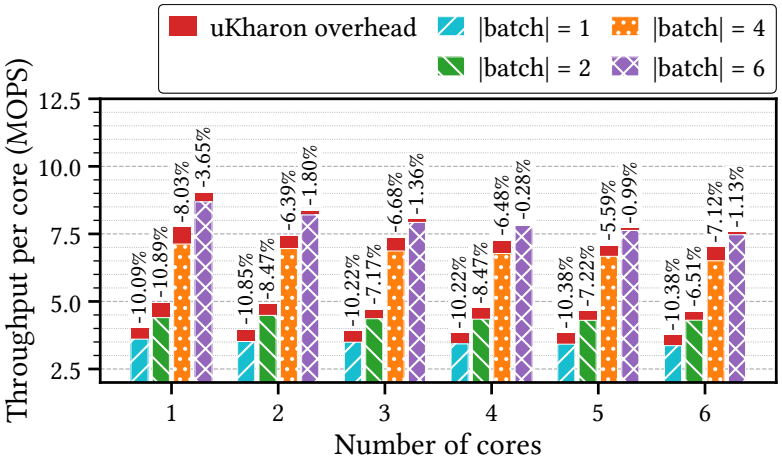

Figure 3.4: Impact of uKharon on HERD's throughput for different batch sizes and numbers of cores. Full bar shows the throughput w/o uKharon; labels show uKharon's overhead.

In fact, RPC-based renewal requires at most 2µs longer leases (compared with reading from coordinators) to achieve the same percentages of timely lease renewal. We attribute this difference to RPC, which involves the CPU of both the application and the lease cache.

From this experiment we select the lease duration that we use for the rest of our evaluation. We pick the lease duration when renewing from coordinators (δ) to be 23 µs, and the lease duration when renewing from lease caches (Δ) to be 25 µs.

Throughput reduction. We use uKharon to make HERD dynamic. The original HERD assumes a static set of servers, each of which serves a shard of the key space. Clients are aware of this sharding and use the key of a request to determine the appropriate server. The lack of dynamicity affects HERD's flexibility in two ways. First, if a server fails, its shard becomes unavailable forever. Second, the system is unable to re-balance the load among the servers. Importantly, the use of a static set of servers ensures consistency of clients' requests: `GETs` return the value of the most recent `PUT`.

In our implementation, each server dedicates up to 6 cores to the KV-cache and each core is responsible for a part of the key space. Every core processes clients' requests and invokes the `Active` method before replying to avoid inconsistencies. If `Active` returns `true`, the core executes the request (if the key belongs to its shard) and replies to the client. Otherwise, the core rejects the request. Given that every core invokes `Active` in the critical path of serving requests, the latency of requests increases (by ~38 ns) and the throughput decreases.

Figure 3.4 shows the per-core throughput of a static deployment of HERD, along with the drop in performance caused by the integration of the `Active` method. The workload is 80% `GETs` and 20% `PUTs` with 32 B values. We vary the number of cores from 1 up to 6 as well as the batch size (i.e., the number of clients' requests processed at once). Typically, static HERD issues a

reply every 350 ns. Without batching, having `Active` in the critical path raises the reply time to 388 ns, an increase of 11%. Batching has a positive impact on `Active`'s overhead as a single call to the method is used to serve all the requests in a batch. Thus, for batches of 6 replies, `Active` effectively takes 38/6 = 6.3 ns per reply, an increase of just 1.8%. Finally, the overhead of `Active` does not increase with the number of cores, even though they invoke the method concurrently. This indicates good multicore scalability, which implies that a single uKharon Core instance per server is sufficient to serve all applications running on it.

Bandwidth overhead. uKharon Core reduces the bandwidth available to applications. Lease renewal requires 240 B when contacting 3 coordinators and 132 B when contacting a lease cache, which translates to (assuming renewal every 10 µs) 192 Mbps and 105 Mbps, respectively. This bandwidth requirement accounts for 0.1–0.2% of a 100 Gbps link, thus the bandwidth of application nodes is marginally impacted. Failure detection has similar bandwidth requirement.

3.7.2 Failover Time

We study uKharon's failover time considering userspace and kernel failures. We do not further evaluate catastrophic failures, as 95% of the failover is for their 1 ms-long detection, making microsecond-scale agreement and leases insignificant.

Table 3.2 summarizes the median failover (over 100 measurements) for various failure scenarios. We consider the failure of a single application node optionally combined with the failure of the coordinator leader or/and a lease cache. We emulate simultaneous failures by relying on RDMA Multicast. An auxiliary program executes alongside the program which we emulate the failure of. When the auxiliary program receives the multicast message, it uses `SIGKILL` to kill the targeted program. We assume the worst scenario, i.e., the failure of the application node results in global unavailability that is resolved only by a new (active) membership that excludes it.

In every entry of Table 3.2, we present the failover time when detecting the failure using the deadbeat mechanism (left) and the RDMA-based heartbeat mechanism (right). We now discuss the failover time when using the deadbeat, first considering the case when the lease caches are absent. For a single application failure, uKharon is able to failover in 50 µs using the deadbeat. If the coordinator leader crashes at the same time as the application, the failover time increases by around 15 µs. We attribute this increase to (1) the leader switch mechanism of the consensus engine (~10 µs) and (2) the imperfect synchronization of `SIGKILL` among the failed nodes (~5 µs). When lease caches are part of uKharon, the failover times for the same failure scenarios increase (as expected) by 20–25 µs, which is about the lease duration of the cache. Failure of a cache has no impact on the failover time (bottom entries of the first and third columns). This is because (1) the application node receives the broadcast failure notification and switches lease cache before the membership changes and (2) the new

L exists?	**A**	**A** + **C**	**A** + **L**	**A** + **L** + **C**
No	50\96	64\114	-	-
Yes	74\108	96\138	75\113	101\139

Table 3.2: Failover time (in µs) for failures in **A**pp, **C**oordinator leader and **L**ease caches; using the deadbeat\heartbeat.

membership is compatible with the previous one. The simultaneous failure of all three types of nodes has a downtime of 101 µs, instead of 96 µs. Again, the failure of the cache does not affect the failover time, but with three nodes the imperfect synchronization of failures adds up. Finally, the same failures when using the RDMA-based heartbeat mechanism range from 96 to 139 µs. This mechanism adds ~45 µs of failover compared to the deadbeat. The reason is that reading the same value twice upon failure takes 1.5 delays on expectation and READs are issued every 30 µs.

3.7.3 uKharon-KV

Both uKharon-KV and HERD+Mu follow a primary-backup replication scheme. All requests are served by the primary, which replicates them to backups. Backups are only used for fault tolerance. All replicas (primary and backups) execute requests in the same order, but only the primary replies to clients. In the event of a failure of the primary, one of the backups becomes the new primary and continues serving clients' requests. All replicas execute all requests in the same total order, thus replicas are an exact copy of the failed primary. This means that when a replica becomes the new primary, it can respond to clients without breaking consistency.

One problem these systems have to deal with is multiple nodes trying to replicate clients' requests simultaneously. This happens when the primary fails and multiple nodes, believing they are the new primary, try to handle clients' requests. Mu avoids this problem by relying on RDMA permissions (see §3.5.3). On the other hand, uKharon-KV relies exclusively on the membership service to address it. Each membership determines a single primary. When the primary fails, a new membership is emitted that determines the new primary. Since only one membership is active at a time, no two replicas can believe to be the primary simultaneously.

The replication protocol of uKharon-KV works as follows: The primary P replicates all clients' requests to a single backup B by RMDA-writing them to a dedicated buffer on the latter. In parallel, P *speculatively* executes the requests. Upon completion of the RDMA WRITE, the primary checks that the membership in which P is the primary is still active. If that is the case, P replies to the client. Otherwise, P drops the request. Upon membership change, B waits for the new membership—in which it is the primary—to become active. Then, B scans the local buffer that was dedicated to P and applies all unprocessed requests in it. Only then B starts processing clients' requests. The client's failover time is the time interval between the client's last successful request to P and its first successful request to B (as the new primary).

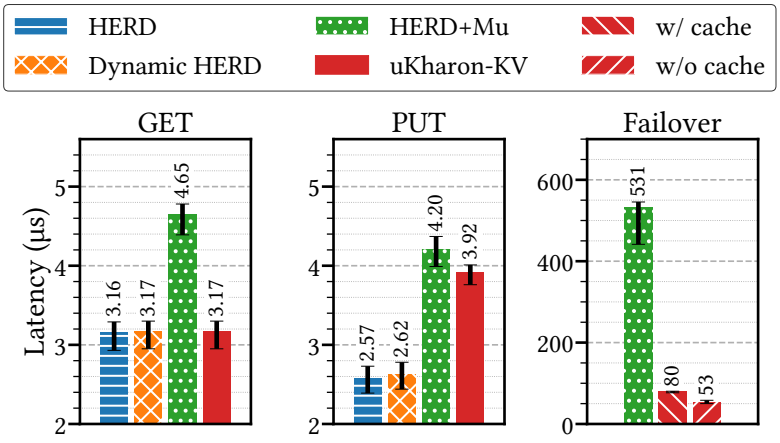

Figure 3.5: Latency comparison (left) of vanilla HERD, Dynamic HERD, HERD+Mu, uKharon-KV. Failover time comparison (right) of HERD+Mu and uKharon-KV. HERD+Mu uses 3-way replication; uKharon-KV uses its deadbeat. Bar height shows 95th %-ile latency; numerical label shows the 95th %-ile; error bars show the median and 99th %-ile.

If *P*'s speculative execution turns out to be incorrect, its state may diverge from the one of the new primary *B*. uKharon-KV, however, does not follow the common practice of rolling back unsuccessful speculations, because our prototype adopts a simple design: when a node is removed from the membership, it is not allowed to re-enter the system. Thus, the state of the old primary *P* is no longer used when *B* takes over, hence skipping the rollback.

Replication latency. We compare the latency of HERD, HERD+Mu and uKharon-KV. For HERD, we deploy a single node. For HERD+Mu, we deploy three nodes, a primary and two backups, all of which execute an instance of HERD and Mu. For uKharon-KV, we deploy a primary and a backup, both running uKharon-KV, as well as three coordinators. For these experiments, a HERD client connects to the primary and issues `PUT` and `GET` requests. We measure the time it takes for a client to complete a request and compute the median, the 95th and the 99th percentiles over 10 million requests.

Figure 3.5 shows the end-to-end latency of vanilla HERD and of both replication approaches. In vanilla HERD, `PUT`s are more efficient than `GET`s by 23%, due to the way HERD handles the two types of requests. Briefly, `PUT`s rely mostly on RDMA WRITEs, which is the most efficient RDMA verb [92], while `GET`s rely mostly on RDMA SENDs. For reference, we also show the latency of Dynamic HERD, which uses uKharon's `Active` method in the critical path of executing clients' requests, as explained in section 3.7.1. We verify, once again, the efficiency of the `Active` method. At the 95th percentile, Dynamic HERD's requests are delayed by 10 ns (for `GET`s) and 50 ns (for `PUT`s), compared with vanilla HERD.

The two replicated solutions exhibit different costs. HERD+Mu replicates all requests, regardless of whether they are `PUT`s or `GET`s, while uKharon-KV replicates only `PUT`s. HERD+Mu

does not distinguish between `PUT`s and `GET`s, because in Mu the primary uses the result of replication (whether it is successful or not) to determine if it is still the primary or not. If Mu were to skip the replication of `GET`s, inconsistency would occur (see §3.2.1). On the other hand, uKharon-KV executes `GET`s locally, without replicating them, since the primary relies on the `Active` method to determine if its data is stale or not. Also, observe that uKharon-KV replicates `PUT`s approximately 300 ns faster than Mu. This improvement is merely attributed to the speculative approach adopted by uKharon-KV. In HERD+Mu, the primary executes the request *after* it has been replicated to a majority. On the other hand, the primary in uKharon-KV executes the request in *parallel* to the replication to the backup. Thus, our solution hides the cost of executing the request, which is approximately 300 ns, as shown by the difference of the two rightmost bars in the middle plot of Figure 3.5. Regardless, uKharon-KV provides the same fault tolerance as Mu, even with one less replica: if a single replica crashes in either HERD+Mu or uKharon-KV, the system remains operational but cannot tolerate another failure. Fundamentally, both HERD+Mu and uKharon-KV assume a majority of correct nodes, the former among the replicas and the latter among the coordinators.

Failover. We compare the failover latency of uKharon-KV with HERD+Mu in the event of userspace failures. We run uKharon-KV in two configurations. In the first one, clients directly RDMA-read from coordinators to renew their lease. In the second one, clients go to lease caches. The third graph of Figure 3.5 shows that HERD+Mu has a 95th-percentile failover time of 531 µs. This number is almost half of what Mu's authors report since we fine-tuned their failure detector for our own setup. At the same time, uKharon-KV without cache (resp. with) achieves a 10× improvement (resp. 6.5×) at 53 µs (resp. 80 µs) of end-to-end failover time.

3.8 Related Work

Membership services in general. Membership services are widely used in the data center. Distributed data processing applications (e.g., Kafka [103], MapReduce [55]), storage systems (e.g., Cassandra [105], HDFS [161]) and orchestration tools (e.g., Mesos [81]) rely on Zookeeper [84] for leader election, membership management, locks, watches, etc. uKharon focuses on membership management, yet it can be extended to support Zookeeper's features. Indeed, uKharon-KV (excluding the lack of durability) offers similar guarantees to the strongly consistent KV-store of Zookeeper, which comprises its basic building block. For instance, locks can be implemented on top of uKharon-KV by extending its interface with CompareAndSwap. Watches, being an unreplicated pub/sub system, only require modifying uKharon-KV's primary. The important difference is that Zookeeper is not suitable for the microsecond scale and does not exploit RDMA.

Failure detection in the data center. A common approach to detect failures is to use end-to-end timeouts, which are hard to set. Falcon [119] proposes to use inside information in order

to build faster and more accurate failure detectors by relying on hierarchies of specialized detectors. It maximizes accuracy by killing suspected processes. Albatross [118] is slightly more forgiving and isolates suspected processes so that they cannot affect the state of the system. Pigeon [117] provides fine-grained reports that end applications use to act accordingly. We embrace Falcon's philosophy and use RDMA-tailored failure detectors to operate at the microsecond scale.

Time-bound leases. Time-bound leases are widely used to implement consistent distributed applications at the price of some synchrony assumptions. They are often provided by a distributed coordination framework such as ZooKeeper [84] or etcd [146]. Leases are used for leader election [164], as well as for guarding memberships (e.g., in FaRM [64] and Hermes [94]). uKharon guards memberships with purely client-side leases. As a result, uKharon brings leases down to a few tens of microseconds and only assumes bounded clock drift instead of loosely synchronized clocks as in Hermes.

Part III

Microsecond-Scale Byzantine Fault-Tolerant Replication

4 Frugal Byzantine Computing

In Part III of this thesis, starting with this chapter, we turn our attention to Byzantine Fault-Tolerant (BFT) replication of microsecond-scale applications. Yet, from early on we realize that traditional techniques for handling Byzantine failures are expensive: digital signatures are too costly, while using $3f+1$ replicas is uneconomical (f denotes the maximum number of Byzantine processes). We devise algorithms for the classical problems of broadcast and consensus, which constitute the foundational components of replication protocols. The algorithms reduce the number of replicas to $2f+1$ and minimize the number of signatures. To achieve the first goal, we rely on the message-and-memory model [3], a theoretical model that describes the RDMA technology. However, even in this model, accomplishing the second goal simultaneously is challenging.

We first address this challenge for the problem of broadcasting messages reliably. We study two variants of this problem, Consistent Broadcast and Reliable Broadcast, typically considered very close. Perhaps surprisingly, we establish a separation between them in terms of signatures required. In particular, we show that Consistent Broadcast requires at least 1 signature in some execution, while Reliable Broadcast requires $O(n)$ signatures in some execution. We present matching upper bounds for both primitives within constant factors.

We then turn to the problem of consensus and argue that this separation matters for solving consensus with Byzantine failures: we present a consensus algorithm that uses Consistent Broadcast as its main communication primitive. This algorithm works for $n = 2f+1$ and avoids signatures in the common case—properties that have not been simultaneously achieved previously and that pave the way towards microsecond-scale BFT replication.

4.1 Introduction

Byzantine fault-tolerant computing is typically associated with high cost. To tolerate f failures, we typically need $n = 3f + 1$ replica processes. Moreover, the agreement protocols for synchronizing the replicas have a significant latency overhead. Part of the overhead comes

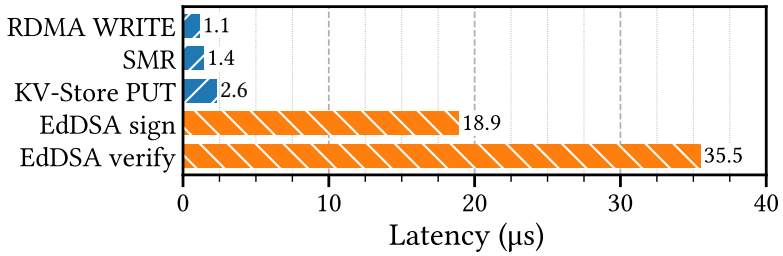

Figure 4.1: Latency of writing 32 B over RDMA, replicating a request on Mu [5] (SMR), executing a PUT on the HERD key-value cache [91], and signing or verifying a message using state-of-the-art EdDSA signatures [89].

from network delays, but digital signatures—often used in Byzantine computing—are even more costly than network delays. For instance, signing or verifying a message can be 17–32 times slower than sending it over a low-latency Infiniband fabric (Figure 4.1).

In this chapter, we study whether Byzantine computing can be *frugal*, meaning if it can use few processes and few signatures. By Byzantine computing, we mean the classical problems of broadcast and consensus. By frugality, we first mean systems with $n = 2f + 1$ processes, where f is the maximum number of Byzantine processes. Such systems are clearly preferable to systems with $n = 3f + 1$, as they require 33–50% less hardware. However, seminal impossibility results imply that in the standard message-passing model with $n = 2f + 1$ processes, neither consensus nor various forms of broadcast can be solved, even under partial synchrony or randomization [66]. To circumvent the above impossibility results, we consider a message-and-memory (M&M) model, which allows processes to both pass messages and share memory, capturing the latest hardware capabilities of enterprise servers [3, 4]. In this model, it is possible to solve consensus with $n = 2f + 1$ processes and partial synchrony [4].

Frugality for us also means the ability to achieve *low latency*, by minimizing the number of digital signatures used. Mitigating the cost of digital signatures is commonly done by replacing them with more computationally efficient schemes, such as message authentication codes (MACs). For instance, with $n = 3f + 1$, the classic PBFT replaces some of its signatures with MACs [38], while Bracha's broadcast algorithm [29] relies exclusively on MACs. As we show, when $n = 2f + 1$, the same signature-saving techniques are no longer applicable.

The two goals—achieving high failure resilience while minimizing the number of signatures—prove challenging when combined. Intuitively, this is because with $n = 2f + 1$ processes, two quorums may intersect only at a Byzantine process; this is not the case with $n = 3f + 1$. Thus, we cannot rely on quorum intersection alone to ensure correctness; we must instead restrict the behavior of Byzantine processes to prevent them from providing inconsistent information to different quorums. Signatures can restrict Byzantine processes from lying, but only if there are enough correct processes to exchange messages and cross-check information. The challenge is to make processes prove that they behave correctly, based on the information they received so far, while using as few signatures as possible.

We focus initially on the problem of broadcasting a message reliably—one of the simplest and most widely used primitives in distributed computing. Here, a designated sender process s would like to send a message to other processes, such that all correct processes deliver the same message. The difficulty is that a Byzantine sender may try to fool correct processes to deliver different messages. Both broadcast variants, Consistent and Reliable Broadcast, ensure that (1) if the sender is correct, then all correct processes deliver its message, and (2) any two correct processes that deliver a message must deliver the *same* message. Reliable Broadcast ensures an additional property: if any correct process delivers a message, then all correct processes deliver that message.

Perhaps surprisingly, in the M&M model we show a large separation between the two broadcasts in terms of the number of signatures (by correct processes) they require. We introduce a special form of indistinguishability argument for $n = 2f + 1$ processes that uses signatures and shared memory in an elaborate way. With it, we prove lower bounds for deterministic algorithms. For Consistent Broadcast, we prove that any solution requires one correct process to sign in some execution, and provide an algorithm that matches this bound. In contrast, for Reliable Broadcast, we show that any solution requires at least $n - f - 2$ correct processes to sign in some execution. We provide an algorithm for Reliable Broadcast based on our Consistent Broadcast algorithm which follows the well-known Init-Echo-Ready pattern [29] and uses up to $n + 1$ signatures, matching the lower bound within a factor of 2.

To lower the impact of signatures on the latency of our broadcast algorithms, we introduce the technique of background signatures. Given the impossibility of completely eliminating signatures, we design our protocols such that signatures are not used in well-behaved executions, i.e., when processes are correct and participate within some timeout. In other words, both broadcast algorithms generate signatures in the background and also incorporate a fast path where signatures are not used.

We next show how to use our Consistent Broadcast algorithm to improve consensus algorithms. The algorithm is based on PBFT [37], and maintains *views* in which one process is the *primary*. Within a view, agreement can be reached by simply having the primary consistent-broadcast a value, and each replicator respond with a consistent broadcast. When changing views, a total of $O(n^2)$ calls to Consistent Broadcast may be issued. The construction within a view is similar to our Reliable Broadcast algorithm. Interestingly, replacing this part with the Reliable Broadcast abstraction does *not* yield a correct algorithm; the stronger abstraction hides information that an implementation based on Consistent Broadcast can leverage. For the correctness of our algorithm, we rely on a technique called *history validation* and on *cross-validating* the view-change message. Our consensus algorithm has four features: (1) it works for $n = 2f + 1$ processes, (2) it issues no signatures on the fast path, (3) it issues $O(n^2)$ signatures on a view-change and (4) it issues $O(n)$ background signatures within a view. As far as we know, no other algorithm achieves all these features simultaneously. This result provides a strong motivation for the use of Consistent Broadcast—rather than Reliable Broadcast—as a first-class primitive in the design of agreement algorithms.

To summarize, we quantify the impossibility of avoiding signatures by proving lower bounds on the number of signatures required to solve the two variants of the broadcast problem—*Consistent* and *Reliable Broadcast*—and provide algorithms that match our lower bounds. Also, we construct a consensus algorithm using the Consistent Broadcast primitive. For our analysis, we consider the message-and-memory model [3, 4], but our results also apply to the pure shared memory model: our algorithms do not require messages so they work under shared memory, while our lower bounds apply *a fortiori* to shared memory.

4.2 Related Work

Message-and-memory models. We adopt a message-and-memory (M&M) model, which is a generalization of both message-passing and shared-memory. M&M is motivated by enterprise servers with the latest hardware capabilities—such as RDMA, RoCE, Gen-Z, and soon CXL—which allow machines to *both* pass messages and share memory. M&M was introduced by Aguilera et al. in [3], and subsequently studied in several other works [4, 13, 79, 150]. Most of these works did not study Byzantine fault tolerance, but focused on crash-tolerant constructions when memory is shared only by subsets of processes [3, 13, 79, 150]. In [4], Aguilera et al. consider crash- and Byzantine- fault tolerance, as well as bounds on communication rounds on the fast path for a variant of the M&M model with dynamic access permissions and memory failures. However, they did not study any complexity bounds off the fast path, and in particular did not consider the number of signatures such algorithms require.

Byzantine fault tolerance. Lamport, Shostak and Pease [113, 143] show that Byzantine agreement can be solved in synchronous message-passing systems iff $n \geq 3f + 1$. In asynchronous systems subject to failures, consensus cannot be solved [70]. However, this result is circumvented by making additional assumptions for liveness, such as randomization [23, 135] or partial synchrony [42, 66]. Even with signatures, asynchronous Byzantine agreement can be solved in message-passing systems only if $n \geq 3f + 1$ [31]. Dolev and Reischuk [62] prove a lower bound of $n(f + 1)/4$ signatures for Byzantine agreement, assuming that every message carries at least the signature of its sender.

Byzantine broadcast. In the message-passing model, both Consistent and Reliable Broadcast require $n \geq 3f + 1$ processes, unless (1) the system is synchronous and (2) digital signatures are available [31, 61, 163]. Consistent Broadcast is sometimes called Crusader Agreement [61]. The Consistent Broadcast abstraction was used implicitly in early papers on Byzantine broadcast [30, 169], but its name was coined later by Cachin et al. in [35]. The name "consistent broadcast" may also refer to a similar primitive used in synchronous systems [124, 163]. Our Reliable Broadcast algorithm shares Bracha's Init-Echo-Ready structure [29] with other broadcast algorithms [31, 152, 163], but is the first algorithm to use this structure in shared memory to achieve Reliable Broadcast with $n = 2f + 1$ processes.

BFT with stronger communication primitives. Despite the known fault tolerance bounds for asynchronous Byzantine Failure Tolerance (BFT), Byzantine consensus can be solved in asynchronous systems with $2f+1$ processes if stronger communication mechanisms are assumed. Some prior work solves Byzantine consensus with $2f+1$ processes using specialized trusted components that Byzantine processes cannot control [43, 44, 49, 50, 93, 172]. These trusted components can be seen as providing a broadcast primitive for communication. In contrast to us, these works assume the existence of such primitives as black boxes, and do not study the cost of implementing them using weaker hardware guarantees. We achieve the same Byzantine fault-tolerance by using the shared memory to prevent the adversary from partitioning correct processes: once a correct process writes to a register, the adversary cannot prevent another correct process from seeing the written value.

It has been shown that shared memory primitives can be useful in providing BFT if they have *access control lists* or *policies* that dictate the allowable access patterns in an execution [4, 8, 26, 27, 126]. Alon et al. [8] provide tight bounds for the number of strong shared-memory objects needed to solve consensus with optimal resilience. They do not, however, study the number of signatures required.

Early termination. The idea of having a *fast path* that allows early termination in well-behaved executions is not a new one, and has appeared in work on both message-passing [4, 5, 14, 60, 96, 102, 111] and shared-memory [18, 168] systems. Most of these works measure the fast path in terms of the number of message delays (or network rounds trips) they require, but some also consider the number of signatures [14]. In this chapter, we show that a signature-free fast path does not prevent an algorithm from having an optimal number of overall signatures.

4.3 Model and Preliminaries

We consider an asynchronous message-and-memory model, which allows processes to use both message-passing and shared-memory [3]. The system has n processes $\Pi = \{p_1, \ldots, p_n\}$ and a shared *memory* M. Throughout this chapter, the term memory refers to M, not to the local state of processes. We sometimes augment the system with eventual synchrony (§4.3.2).

Communication. The memory consists of single-writer multi-reader (SWMR) read/write atomic registers. Each process can read all registers, and has access to an unlimited supply of registers it can write. If a process p can write to a register r, we say that p *owns* r. This model is a special case of access control lists (ACLs) [126], and of dynamically permissioned memory [4]. Additionally, every pair of processes p and q can send messages to each other over links that satisfy the *integrity* and *no-loss* properties. Integrity requires that a message m from p be received by q at most once and only if m was previously sent by p to q. No-loss requires that a message m sent from p to q be eventually received by q.

Signatures. Our algorithms assume digital signatures: each process can *sign* and *verify* signatures. A process p may sign a value v, producing $\sigma_{p,v}$; when unambiguous, we drop the subscripts. Given v and $\sigma_{p,v}$, a process can verify whether $\sigma_{p,v}$ is a valid signature of v by p.

Failures. Up to f processes may fail by becoming Byzantine, where $n = 2f+1$. Such a process can deviate arbitrarily from the algorithm, but cannot write on a register that is not its own, and cannot forge the signature of a correct process. As usual, Byzantine processes can collude, e.g., by using side-channels to communicate. The memory M does not fail; such a reliable memory is implementable from a collection of fail-prone memories [4]. We assume that these individual memories may only fail by crashing.

4.3.1 Broadcast

We consider two broadcast variants: Consistent Broadcast [34, 35] and Reliable Broadcast [28, 34]. In both variants, broadcast is defined in terms of two primitives: *broadcast*(m) and *deliver*(m). A designated *sender* process s is the only one that can invoke *broadcast*. When s invokes *broadcast*(m) we say that *s broadcasts m*. When a process p invokes *deliver*(m), we say that *p delivers m*.

Definition 1. Consistent Broadcast *has the following properties:*

Validity *If a correct process s broadcasts m, then every correct process eventually delivers m.*

No duplication *Every correct process delivers at most one message.*

Consistency *If p and p' are correct processes, p delivers m, and p' delivers m', then $m{=}m'$.*

Integrity *If some correct process delivers m and s is correct, then s previously broadcast m.*

Definition 2. Reliable Broadcast *has the following properties:*

Validity, No duplication, Consistency, Integrity *Same properties as in Definition 1.*

Totality *If some correct process delivers m, then every correct process eventually delivers a message.*

We remark that both broadcast variants behave the same way when the sender is correct and broadcasts m. However, when the sender is faulty Consistent Broadcast has no delivery guarantees for correct processes, i.e., some correct processes may deliver m, others may not. In contrast, Reliable Broadcast forces every correct process to eventually deliver m as soon as one correct process delivers m.

4.3.2 Consensus

Definition 3. Weak Byzantine agreement [109] *has the following properties:*

Agreement *If correct processes i and j decide val and val', respectively, then $val = val'$.*

Weak validity *If all processes are correct and some process decides val, then val is the input of some process.*

Integrity *No correct process decides twice.*

Termination *Eventually every correct process decides.*

Our consensus algorithm (§4.6) satisfies agreement, validity, and integrity under asynchrony, but requires eventual synchrony for termination. That is, we assume that for each execution there exists a *Global Stabilization Time (GST)*, unknown to the processes, such that from GST onwards there is a known bound Δ on communication and processing delays.

4.4 Lower Bounds on Broadcast Algorithms

We show lower bounds on the number of signatures required to solve Consistent and Reliable Broadcast with $n = 2f + 1$ processes in our model. We focus on signatures by correct processes because Byzantine processes can behave arbitrarily (including signing in any execution).

4.4.1 High-Level Approach

Broadly, we use indistinguishability arguments that create executions E_v and E_w that deliver different messages v and w; then we create a composite execution E where a correct process cannot distinguish E from E_v, while another correct process cannot distinguish E from E_w, so they deliver different values, a contradiction. Such arguments are common in message-passing system, where the adversary can prevent communication by delaying messages between correct processes. However, it is not obvious how to construct this argument in shared memory, as the adversary cannot prevent communication via the shared memory, especially when using single-writer registers that cannot be overwritten by the adversary. Specifically, if correct processes write their values and read all registers, then for any two correct processes, at least one sees the value written by the other [22]. So, when creating execution E in which, say E_v occurs first, processes executing E_w will know that others executed E_v beforehand.

We handle this complication in two ways, depending on whether the sender signs its broadcast message. If the sender does not sign, we argue that processes executing E_w cannot tell whether E_v was executed by correct or Byzantine processes, and must therefore still output their original value w. This is the approach in the lower bound proof for Consistent Broadcast (Lemma 4.4.1).

However, once a signature is produced, processes can save it in their memory to prove to others that they observed a valid signature. Thus, if the sender signs its value, then processes executing E_w cannot be easily fooled; if they see two different values signed by the sender, then the sender is provably faulty, and correct processes can choose a different output. So, we need another way to get indistinguishable executions. We rely on a *correct bystander* process. We make a correct process b in E sleep until all other correct processes decide. Then b wakes up and observes that E is a composition of E_v and E_w. While b can recognize that E_v or E_w was executed by Byzantine processes, it cannot distinguish which one. So, b cannot reliably output the same value as other correct processes. We use this construction for Reliable Broadcast, but we believe it applies to other agreement problems in which all correct processes must decide.

The proof is still not immediate from here. In particular, since $f<n/2$, correct processes can wait until at least $f+1$ processes participate in each of E_v and E_w. Of those, in our proof we assume at most $f-1$ processes sign values. Since we need a bystander later, only $2f$ processes can participate. Thus, the sets executing E_v and E_w overlap at two processes; one must be the sender, to force decisions in both executions. Let p be the other process and S_v and S_w be the set that execute E_v and E_w respectively, without the sender and p. Thus, $|S_v| = |S_w| = f-1$.

The key complication is that if p signs its values in one of these two executions, we cannot compose them into an execution E in which the bystander b cannot distinguish which value it should decide. To see this, assume without loss of generality that p signs a value in execution E_w. To create E, we need the sender s and the set S_w to be Byzantine. The sender will produce signed versions of both v and w for the two sets to use, and S_w will pretend to execute E_w even though they observed that E_v was executed first. Since $|S_w|+|\{s\}| = f$, all other processes must be correct. In particular, p will be correct, and will not produce the signature that it produces in E_w. Thus, the bystander b will know that S_v were correct. More generally, the problem is that, while we know that at most $f-1$ processes sign, we do not know *which* processes sign. A clever algorithm can choose signing processes to defeat the indistinguishability argument—in our case, this happens if p is a process that signs.

Due to this issue, we take a slightly different approach for the Reliable Broadcast lower bound, first using the bystander construction to show that any Reliable Broadcast algorithm must produce *a single non-sender* signature. To strengthen this to our bound, we construct an execution in which this signature needs to be repeatedly produced. To make this approach work, we show not just that *there exists* an execution in which a non-sender signature is produced, but that *for all* executions of a certain form, a non-sender signature is produced. This change in quantifiers requires care in the indistinguishability proof, and allows us to repeatedly apply the result to construct a single execution that produces many signatures.

4.4.2 Proofs

In all proofs in this section, we denote by s the designated sender process in the broadcast protocols we consider. We first show that Consistent Broadcast requires at least one signature.

Lemma 4.4.1. *Any algorithm for Consistent Broadcast in the M&M model with $n = 2f + 1$ and $f \geq 1$ has an execution in which at least one correct process signs.*

Proof. By contradiction, assume there is some algorithm A for Consistent Broadcast in the M&M model with $n = 2f + 1$ and $f \geq 1$ without any correct process signing. Partition Π into 3 subsets: S_1, S_2, and $\{p\}$, where S_1 contains the sender, $|S_1| = f$, $|S_2| = f$, and p is a single process. Let v, w be two distinct messages. Consider the following executions.

EXECUTION $E_{\text{CLEAN-V}}$. Processes in S_1 and p are correct (including the sender s), while processes in S_2 are faulty and never take a step. Initially, s broadcasts v. Since s is correct, processes in S_1 and p eventually deliver v. By our assumption that correct processes never sign, processes in S_1 and p do not sign in this execution; processes in S_2 do not sign either, because they do not take any steps.

EXECUTION $E_{\text{DIRTY-W}}$. Processes in S_1 and S_2 are correct but p is Byzantine. Initially, p sends all messages and writes to shared memory as it did in $E_{\text{CLEAN-V}}$ (it does so without following its algorithm; p is able to do this since no process signed in $E_{\text{CLEAN-V}}$). Then, the correct sender s broadcasts w and processes in S_1 and S_2 execute normally, while p stops executing. Then, by correctness of the algorithm, eventually all correct processes deliver w. By our assumption that correct processes never sign, processes in S_1 and S_2 do not sign in this execution; p does not sign either, because it acts as it did in $E_{\text{CLEAN-V}}$.

EXECUTION E_{BAD}. Processes in S_1 are Byzantine, while processes in S_2 and p are correct. Initially, processes in S_2 sleep, while processes in S_1 and p execute, where processes in S_1 send the same messages to p and write the same values to shared memory as in $E_{\text{CLEAN-V}}$ (but they do not send any messages to S_2), so that from p's perspective the execution is indistinguishable from $E_{\text{CLEAN-V}}$. S_1 are able to do this because no process signed in $E_{\text{CLEAN-V}}$. Therefore, p eventually delivers v. Next, processes in S_1 write the initial values to their registers[1]. Now, process p stops executing, while processes in S_1 and S_2 execute the same steps as in $E_{\text{DIRTY-W}}$—here, note that S_2 just follows algorithm A while S_1 is Byzantine and pretends to be in an execution where s broadcasts w (S_1 is able to do this because no process signed in $E_{\text{DIRTY-W}}$). Because this execution is indistinguishable from $E_{\text{DIRTY-W}}$ to processes in S_2, they eventually deliver w. At this point, correct process p has delivered v while processes in S_2 (which are correct) have delivered w, which contradicts the consistency property of Consistent Broadcast. □

An algorithm for Reliable Broadcast works for Consistent Broadcast, so Lemma 4.4.1 also applies to Reliable Broadcast.

We now show a separation between Consistent Broadcast and Reliable Broadcast: any algorithm for Reliable Broadcast has an execution where at least $f-1$ correct processes sign.

[1] Recall that registers are single-writer. By "their registers", we mean the registers to which the processes can write.

The proof for the Reliable Broadcast lower bound has two parts. First, we show that intuitively there are many executions in which some process produces a signature: if E is an execution in which (1) two processes never take steps, (2) the sender is correct, and (3) processes fail only by crashing, then some non-sender process signs. This is the heart of the proof, and relies on the indistinguishability arguments discussed in Section 4.4.1. Here, we focus only on algorithms in which at most f correct processes sign, otherwise the algorithm trivially satisfies our final theorem.

Lemma 4.4.2. *Let A be an algorithm for Reliable Broadcast in the M&M model with $n = 2f + 1$ and $f \geq 2$ processes, such that in any execution at most f correct processes sign. In all executions of A in which at least 2 processes crash initially, processes fail only by crashing, and the sender is correct, at least one correct non-sender process signs.*

Proof. By contradiction, assume some algorithm A satisfies the conditions of the lemma, but there is some execution of A where the sender s is correct, processes fail only by crashing, and at least 2 processes crash initially, but no correct non-sender process signs. Let $E_{\text{CLEAN-V}}$ be such an execution, D be a set with two processes that crash initially in $E_{\text{CLEAN-V}}$[2], $C = \Pi \setminus D$, and v be the message broadcast by s in $E_{\text{CLEAN-V}}$. Consider the following executions:

EXECUTION $E_{\text{CLEAN-W}}$. The sender s broadcasts some message $w \neq v$, D crashes initially, and C is correct. Since s is correct, eventually all correct processes deliver w. By assumption, at most f processes sign. Let $S \subset C$ contain all processes that sign, augmented with any other processes so that $|S| = f$. Let $T = C \setminus S$. Note that (1) $|T| = f - 1$ and (2) if s signed, then $s \in S$, otherwise $s \in T$.

EXECUTION $E_{\text{CLEAN-V}}$. This execution was defined above (where s broadcasts v). Since s is correct, eventually all correct processes deliver v. At least one process in T is correct—call it p_t—since processes in D are faulty and there are at least $f + 1$ correct processes. Note that p_t delivers v. We refer to p_t in the next execution.

EXECUTION $E_{\text{MIXED-V}}$. Processes in S are Byzantine and the rest are correct. Initially, the execution is identical to $E_{\text{CLEAN-V}}$, except that (1) processes in D are just sleeping not crashed, and (2) processes in S do not send messages to processes in D (this is possible because processes in S are Byzantine). The execution continues as in $E_{\text{CLEAN-V}}$ until p_t delivers v. Then, processes in S misbehave (they are Byzantine) and do three things: (1) they change their states to what they were at the end of $E_{\text{CLEAN-W}}$ (this is possible because no process in T signed in $E_{\text{CLEAN-W}}$), (2) they write to their registers in shared memory the same last values that they wrote in $E_{\text{CLEAN-W}}$, and (3) they send the same messages they did in $E_{\text{CLEAN-W}}$. Intuitively, processes in S pretend that s broadcast w. Let t be the time at this point; we refer to time t in the next execution. Now, we pause processes in S and let all other processes execute, including D which had been sleeping. Since p_t delivered v and processes in D are correct, they eventually deliver v as well.

[2] If more than two processes crashed initially, pick any two arbitrarily.

EXECUTION E_{BAD}. Processes in $T \cup \{s\}$ are Byzantine and the rest are correct. Initially, the execution is identical to $E_{\text{CLEAN-W}}$, except that (1) processes in D are sleeping not crashed, and (2) processes in $T \cup \{s\}$ do not send messages to processes in D. Execution continues as in $E_{\text{CLEAN-W}}$ until processes in S (which are correct) deliver w. Then, processes in $T \cup \{s\}$ misbehave and do three things: (1) they change their states to what they were in $E_{\text{MIXED-V}}$ at time t—this is possible because in $E_{\text{CLEAN-V}}$ (and therefore in all values and messages they had by time t in $E_{\text{MIXED-V}}$), no non-sender process signed, and in particular, there were no signatures by any process in $S \setminus \{s\}$; (2) they write to the registers in shared memory the same values that they have in $E_{\text{MIXED-V}}$ at time t; and (3) they send all messages they did in $E_{\text{MIXED-V}}$ up to time t. Intuitively, processes in $T \cup \{s\}$ pretend that s broadcast v. Now, processes in D start executing. In fact, execution continues as in $E_{\text{MIXED-V}}$ from time t onward, where processes is S are paused and all other processes execute (including D). Because these processes cannot distinguish the execution from $E_{\text{MIXED-V}}$, eventually they deliver v. Note that processes in D are correct and they deliver v, while processes in S are also correct and deliver w—contradiction. □

In the final stage of the proof, we leverage Lemma 4.4.2 to construct an execution in which many processes sign. This is done by allowing some process to be poised to sign, and then pausing it and letting a new process start executing. Thus, we apply Lemma 4.4.2 $f-1$ times to incrementally build an execution in which $f-1$ correct processes sign.

Theorem 4.4.3. *Any algorithm that solves Reliable Broadcast in the M&M model with $n = 2f+1$, $f \geq 1$ has an execution in which at least $f-1$ correct non-sender processes sign.*

Proof. If $f = 1$, the result is trivial; it requires $f - 1 = 0$ processes to sign.

Now consider the case $f \geq 2$. If A has an execution in which at least $f+1$ correct processes sign, then we are done. Now suppose A has no execution in which at least $f+1$ correct processes sign. Consider the following execution of A.

All processes and s are correct. Initially, s broadcasts v. Then processes $s, p_1 \dots p_f$ participate, and the rest are delayed. This execution is indistinguishable to $s, p_1 \dots p_f$ from one in which the rest of the processes crashed. Therefore, by Lemma 4.4.2, some process in $p_1 \dots p_f$ eventually signs. Call p_1 the first process that signs. We continue the execution until p_1's next step is to make its signature visible. Then, we pause p_1, and let p_{f+1} begin executing. Again, this execution is indistinguishable to $s, p_2 \dots p_{f+1}$ from one in which the rest of the processes crashed, so by Lemma 4.4.2, eventually some process in $p_2 \dots p_{f+1}$ creates a signature and makes it visible. We let the first process to do so reach the state in which it is about to make its signature visible, and then pause it, and let p_{f+2} start executing.

We continue in this way, each time pausing p_i as it is about to make its signature visible, and letting p_{f+i} begin executing. We can apply Lemma 4.4.2 as long as two processes have not participated yet. At that point, $f-1$ processes are poised to make their signatures visible. We

then let these $f-1$ processes each take one step. This yields an execution of A in which $f-1$ correct non-sender processes sign. □

4.5 Broadcast Algorithms

In this section we present solutions for Consistent and Reliable Broadcast. We first implement Consistent Broadcast in Section 4.5.1; then we use it as a building block to implement Reliable Broadcast, in Section 4.5.2. We prove the correctness of our algorithms in Appendix B.1 and B.2. For both algorithms, we first describe the general execution outside the common case, which captures behavior in the worst executions; we then describe how delivery happens fast in the common case (without signatures).

Process roles in broadcast. We distinguish between three process roles in our algorithms: sender, receiver, and replicator. This is similar in spirit to the proposer-acceptor-learner model used by Paxos [110]. Any process may play any number of roles; if all processes play all three roles, then this becomes the standard model. The sender calls *broadcast*, the receivers call *deliver*, and the replicators help guarantee the properties of broadcast. By separating replicators (often servers) from senders and receivers (often clients or other servers), we improve the practicality of the algorithms: clients, by not fulfilling the replicator role, need not remain connected to disseminate information from other clients. Unless otherwise specified, n and f refer only to replicators; independently, the sender and any number of receivers can also be Byzantine. Receivers cannot send or write any values, as opposed to the sender and replicators, but they can read the shared memory and receive messages.

Background signatures. Our broadcast algorithms produce signatures in the background. We do so to allow the algorithms to be signature-free in the common case. Indeed, in the common case, receivers can deliver a message without waiting for background signatures. However, outside the common case, these signatures must still be produced by the broadcast algorithms in case some replicators are faulty or delayed. Both algorithms require a number of signatures that matches the bounds in Section 4.4 within constant factors.

4.5.1 Consistent Broadcast

We give an algorithm for Consistent Broadcast that issues no signatures in the common case, when there is synchrony and no replicator is faulty. Outside this case, only the sender signs.

Algorithm 4.1 shows the pseudocode. The broadcast and deliver events are called *cb-broadcast* and *cb-deliver*, to distinguish them from *rb-broadcast* and *rb-deliver* of Reliable Broadcast. Processes communicate by sharing an array of *slots*: process i can write to *slots[i]*, and can read from all slots. To refer to its own slot, a processes uses index *me*. The sender s uses its slot to broadcast its message while replicators use their slot to replicate the message. Every

slot has two sub-slots—each a SWMR atomic register—one for a message (*msg*) and one for a signature (*sgn*).

To broadcast a message m, the sender s writes m to its *msg* sub-slot (line 6). Then, in the background, s computes its signature for m and writes it to its *sgn* sub-slot (line 9). The presence of *msg* and *sgn* sub-slots allow the sender to perform the signature computation in the background. Sender s can return from the broadcast while this background task executes.

The role of a correct replicator is to copy the sender's message m and signature σ, provided σ is valid. The copying of m and σ (lines 12–19) are independent events, since a signature may be appended in the background, i.e., later than the message. The fast way to perform a delivery does not require the presence of signatures. Note that correct replicators can have mismatching values only when s is Byzantine and overwrites its memory.

A receiver p scans the slots of the replicators. It delivers message m when the content of a majority ($n-f$) of replicator slots contains m and a valid signature by s for m, and no slot contains a different message $m', m' \neq m$ with a valid sender signature (line 28). Slots with sender signatures for $m' \neq m$ result in a no-delivery. This scenario indicates that the sender is Byzantine and is trying to equivocate. Slots with signatures not created by s are ignored so that a Byzantine replicator does not obstruct p from delivering.

When there is synchrony and both the sender and replicators follow the protocol, a receiver delivers without using signatures. Specifically, delivery in the fast path occurs when there is unanimity, i.e., all $n = 2f + 1$ replicators replicated value m (line 25), regardless of whether a signature is provided by s. A correct sender eventually appends σ, and $n - f$ correct replicators eventually copy σ over, allowing another receiver to deliver m via the slow path, even if a replicator misbehaves, e.g., removes or changes its value.

An important detail is the use of a snapshot to read replicators' slots (line 23), as opposed to a simple collect. The scan operation is necessary to ensure that concurrent reads of the replicators' slots do not return views that can cause correct receivers to deliver different messages. To see why, imagine that the scan at line 23 is replaced by a simple collect. Then, an execution is possible in which correct receiver p_1 reads some (correctly signed) message m_1 from $n - f$ slots and finds the remaining slots empty, while another correct receiver p_2 reads $m_2 \neq m_1$ from $n - f$ slots and finds the remaining slots empty. In this execution, p_1 would go on to deliver m_1 and p_2 would go on to deliver m_2, thus breaking the consistency property. We present such an execution in detail in Appendix B.3.

To prevent scenarios where correct receivers see different values at a majority of replicator slots, the *scan* operation works as follows (lines 30–40): first, it performs a collect of the slots. If all the slots are non-empty, then we are done. Otherwise, we re-collect the *empty slots* until no slot becomes non-empty between two consecutive collects. This suffices to avoid the problematic scenario above and to guarantee liveness despite f Byzantine processes.

Algorithm 4.1: Consistent Broadcast Algorithm with sender s

```
1  # Shared:
2  slots - n array of "slots"; each slot is a 2-tuple (msg, sgn) of SWMR atomic
     ↪ registers, initialized to (⊥,⊥).

4  # Sender code:
5  def cb-broadcast(m):
6    slots[me].msg.write(m)
7    In the background:
8      σ = compute signature for m
9      slots[me].sgn.write(σ)

11 # Replicator code:
12 while True:
13   m = slots[s].msg.read()
14   if m != ⊥:
15     slots[me].msg.write(m)
16   sign = slots[s].sgn.read()
17   val = slots[me].msg.read()
18   if val != ⊥ and sign != ⊥ and sign is a valid signature for val:
19     slots[me].sgn.write(sign)

21 # Receiver code:
22 while True:
23   others = scan()
24   if others[i].msg has the same value m for all i in Π: # Fast path
25     cb-deliver(m); break
26   if others contains at least n−f signed copies of the same value m
27     and (∄i: others[i].sgn is a valid signature for others[i].msg and others[i].msg
         ↪ != m):
28     cb-deliver(m); break

30 def scan():
31   others = [slots[i].(msg, sgn).read() for i in Π]
32   done = False
33   while not done:
34     done = True
35     for i in Π:
36       if others[i] == ⊥:
37         others[i] = slots[i].(msg, sgn).read()
38         if others[i] != ⊥:
39           done = False
40   return others
```

4.5.2 Reliable Broadcast

We now give an algorithm for Reliable Broadcast that issues no signatures in the common case, and issues only $n+1$ signatures in the worst case. Algorithm 4.2 shows the pseudocode.

Processes communicate by sharing arrays *Echo* and *Ready*, which have the same structure of sub-slots as *slots* in Section 4.5.1. *Echo[i]* and *Ready[i]* are writable only by replicator i, while the sender s communicates with the replicators using an instance of Consistent Broadcast (CB) and does not access *Echo* or *Ready*. In this CB instance, s invokes *cb-broadcast*, acting as

sender for CB, and the replicators invoke *cb-deliver*, acting as receivers for CB.

To broadcast a message, s *cb-broadcast*s $\langle \text{INIT}, m \rangle$ (line 6). Upon delivering the sender's message $\langle \text{INIT}, m \rangle$, each replicator writes m to its *Echo msg* sub-slot (line 13). Then, in the background, a replicator computes its signature for m and writes it to its *Echo sgn* sub-slot (line 16). By the consistency property of Consistent Broadcast, if two correct replicators r and r' deliver $\langle \text{INIT}, m \rangle$ and $\langle \text{INIT}, m' \rangle$ respectively, from s, then $m = m'$. Essentially, correct replicators have the same value or $\perp$ in their *Echo msg* sub-slot.

Next, replicators populate their *Ready* slots with a *ReadySet*. A replicator r constructs such a *ReadySet* from the $n - f$ signed copies of m read from the *Echo* slots (lines 19–28). In the background, r reads the *Ready* slots of other replicators and copies over—if r has not written one already—any valid *ReadySet* (line 36). Thus, totality is ensured (Definition 2), as the *ReadySet* created by any correct replicator is visible to all correct receivers.

To deliver m, a receiver p reads $n - f$ valid *ReadySet*s for m (line 45).[3] This is necessary to allow a future receiver p' deliver a message as well. Suppose that p delivers m by reading a single valid *ReadySet* R.[4] Then, the following scenario prevents p' from delivering: let sender s be Byzantine and let R be written by a Byzantine replicator r. Moreover, let a *single* correct replicator have *cb-deliver*ed m, while the remaining correct replicators do not deliver at all, which is allowed by the properties of Consistent Broadcast. So, the *ReadySet* contains values from a single correct replicator and f other Byzantine replicators. If r removes R from its *Ready* slot, it will block the delivery for p' since no valid *ReadySet* exists in memory.

A receiver p can also deliver the sender's message m using a fast path. The signature-less fast path occurs when p reads m from the *Echo* slots of all replicators (line 43), and the delivery of the INIT message by the replicators is done via the fast path of Consistent Broadcast. This is the common case, when replicators are not faulty and replicate messages timely. Note that p delivering m via the fast path does not prevent another receiver p' from delivering. Process p' delivers m via the fast path if all the *Echo* slots are in the same state as for p. Otherwise, e.g., some Byzantine replicators overwrite their *Echo* slots, p' delivers m by relying on the $n - f$ correct replicators following the protocol (line 45).

4.6 Consensus

We now give an algorithm for consensus using Consistent Broadcast as its communication primitive, rather than the commonly used primitive, Reliable Broadcast. Our algorithm is based on the PBFT algorithm [37, 38] and proceeds in a sequence of (consecutive) views. It has four features: (1) it works for $n = 2f + 1$ processes, (2) it issues no signatures in the common

[3] In contrast to Algorithm 4.1, receivers need not use the *scan* operation when gathering information from the replicators' *Ready* slots because there can only be a single value with a valid *ReadySet* (Invariant B.2.1).

[4] A similar argument that breaks totality applies if p were to deliver m by reading $n - f$ signed values of m in the replicators' *Echo* slots.

Algorithm 4.2: Reliable Broadcast Algorithm with sender s

```
1  # Shared:
2  Echo, Ready - n array of "slots"; each slot is a 2-tuple (msg, sgn) of SWMR atomic
     ↪ registers, initialized to (⊥,⊥).

4  # Sender code:
5  def rb-broadcast(m):
6    cb-broadcast(⟨INIT,m⟩)

8  # Replicator code:
9  state = WaitForSender # ∈{WaitForSender,WaitForEchos}
10 while True:
11   if state == WaitForSender:
12     if cb-delivered ⟨INIT,m⟩ from s:
13       Echo[me].msg.write(m)
14       In the background:
15         σ = compute signature for m
16         Echo[me].sgn.write(σ)
17       state = WaitForEchos

19   if state == WaitForEchos:
20     ReadySet = ∅
21     for i ∈ Π:
22       other = Echo[i].(msg,sgn).read()
23       if other.msg == m and other.sgn is m validly signed by i:
24         ReadySet.add((i,other))

26     if size(ReadySet) ≥ n − f:
27       ready = True
28       Ready[me].msg.write(ReadySet)

30 In the background:
31   while True
32     if not ready:
33       others = [Ready[i].msg.read() for i in Π]
34       if ∃i: others[i] is a valid ReadySet:
35         ready = True
36         Ready[me].msg.write(others[i])

38 # Receiver code:
39 while True:
40   others = [Echo[i].msg.read() for i in Π]
41   proofs = [Ready[i].msg.read() for i in Π]
42   if others contains n matching values m: # Fast path
43     rb-deliver(m); break
44   if proofs contains n − f valid ReadySet for the same value m:
45     rb-deliver(m); break
```

case, (3) it issues $O(n^2)$ signatures on a view-change and (4) it issues $O(n)$ required background signatures within a view.

Our algorithm uses a sequence of Consistent Broadcast instances indexed by a broadcast sequence number k. When process p broadcasts its k^{th} message m, we say that p broadcasts (k, m). We assume the following ordering across instances, which can be trivially guaranteed:

(**FIFO delivery**) For $k \geq 1$, no correct process delivers (k, m_k) from p unless it has delivered (i, m_i) from p, for all $i < k$.

Algorithm 4.3 shows the pseudocode. Appendix B.3 has its correctness proof. The protocol proceeds in a sequence of consecutive *views*. Each view has a primary process, defined as the view number mod n (line 6). A view has two phases, PREPARE and COMMIT. There is also a view-change procedure initiated by a VIEWCHANGE message.

When a process is the primary (line 9), it broadcasts a PREPARE message with its estimate *init* (line 11), which is either its input value or a value acquired in the previous view (line 10). Upon receiving a valid PREPARE message, a replica broadcasts a COMMIT message (line 20) with the estimate it received in the PREPARE message. We define a PREPARE to be valid when it originates from the primary and either (a) $view = 0$ (any estimate works), or (b) $view > 0$ and the estimate in the PREPARE message has a proof from the previous view. Appendix B.3.1 details the conditions for a message to be valid. When a replica receives an invalid PREPARE message from the primary or times out, it broadcasts a COMMIT message with $\perp$. If a replica accepts a PREPARE message with *val* as estimate and $n - f$ matching COMMIT messages (line 24), it decides on *val*.

The view-change procedure ensures that all correct replicas eventually reach a view with a correct primary and decide. It uses an acknowledgement phase similar to PBFT with MACs [38]. While in [38] the mechanism is used so that the primary can prove the authenticity of a view-change message sent by a faulty replica, we use this scheme to ensure that (a) a faulty participant cannot lie about a committed value in its VIEWCHANGE message and (b) valid VIEWCHANGE messages can be received by all correct replicas.

A replica starts a view-change by broadcasting a signed VIEWCHANGE message with its view-change tuple (line 28). The view-change tuple *(view, val, proof$_{val}$)* is updated when a replica receives a valid PREPARE message (line 15). It represents the last non-empty value a replica accepted as a valid estimate and the view when this occurred. We use the value's proof, *proof$_{val}$*, to prevent a Byzantine replica from lying about its value: suppose a correct replica decides *val* in view v, but in view $v+1$, the primary p is silent, and so no correct replica hears from p; without the proof, a Byzantine replica could claim to have accepted val' in $v+1$ from p during the view-change to $v+1$, thus overriding the decided value *val*.

When a replica receives a valid VIEWCHANGE message, it responds by broadcasting a signed VIEWCHANGEACK containing the VIEWCHANGE message (line 37). A common practice is to send a digest of this message instead of the entire message [37]. We define a VIEWCHANGE message m from p to be valid when the estimate in the view-change tuple corresponds to the value broadcast by p in its latest non-empty COMMIT and m's proof is valid. We point out that, as an optimization, this proof can be removed from the view-change tuple and be provided upon request when required to validate VIEWCHANGE messages. For instance, in the scenario described above, when a (correct) replica r did not accept val' in view $v+1$, as claimed by the Byzantine replica r', r can request r' to provide a proof for val'.

Algorithm 4.3: Consensus protocol based on Consistent Broadcast ($n = 2f + 1$)

```
1  def propose(v_i):
2    view_i = 0; est_i = ⊥; aux_i = ⊥
3    proof_i = ∅; vc_i= (0,⊥,∅)
4    decided_i = False
5    while True:
6      p_i = view_i % n

8      # Phase 1
9      if p_i == i:
10       init_i = est_i if est_i ≠ ⊥ else v_i
11       cb-broadcast(⟨PREPARE, view_i, init_i, proof_i⟩)
12     wait until receive valid ⟨PREPARE, view_i, val, proof⟩ from p_i or timeout on p_i
13     if received valid ⟨PREPARE, view_i, val, proof⟩ from p_i:
14       aux_i = val
15       vc_i = (view_i,val,proof)
16     else:
17       aux_i = ⊥

19     # Phase 2
20     cb-broadcast(⟨COMMIT, view_i, aux_i⟩)
21     wait until receive valid ⟨COMMIT, view_i, *⟩ from n−f processes
22                and (∀j: receive valid ⟨COMMIT, view_i, *⟩ from j or timeout on j)
23     ∀j: R_i[j] = val if received valid ⟨COMMIT, view_i, val⟩ from j else ⊥
24     if ∃val ≠⊥:#_val(R_i)≥ n−f and aux_i == val:
25       try_decide(val)

27     # Phase 3
28     cb-broadcast(⟨VIEWCHANGE, view_i + 1, vc_i⟩_σi)
29     wait until receive n−f non-conflicting view-change certificates for view_i + 1
30     proof_i = set of non-conflicting view-change certificates
31     est_i = val in proof_i associated with the highest view
32     view_i = view_i + 1

34   In the background:
35     when cb-deliver valid ⟨VIEWCHANGE, view', vc⟩_σj from j:
36       # d is the view-change message being ACKed
37       cb-broadcast(⟨VIEWCHANGEACK, d⟩_σi)

39 def try_decide(val):
40   if not decided_i:
41     decided_i = True
42     decide(val)
```

A view-change certificate consists of a VIEWCHANGE message and $n - f - 1$ corresponding VIEWCHANGEACK messages. This way, each view-change certificate has the contribution of at least one correct replica, who either produces the VIEWCHANGE message or validates a VIEWCHANGE message. Thus, when a correct replica r receives a view-change certificate relayed by the primary, r can trust the contents of the certificate.

To move to the next view, a replica must gather a set of $n - f$ non-conflicting view-change certificates Ψ. This step is performed by the primary of the next view, who then includes

this set with its PREPARE message for the new view. Two view-change certificates conflict if their view-change messages carry a tuple with different estimates ($\neq \perp$), valid proof, and same view number. If the set Ψ consists of tuples with estimates from different views, we select the estimate associated with the highest view. Whenever any correct replica decides on a value *val* within a view, the protocol ensures a set of non-conflicting view-change certificates can be constructed only for *val* and hence the value is carried over to the next view(s).

4.6.1 Discussion

We discuss how Algorithm 4.3 achieves the four features mentioned at the beginning of Section 4.6. The first feature (the algorithm solves consensus with $n = 2f + 1$ processes) follows directly from the correctness of the algorithm. The second feature (the algorithm issues no signatures in the common case) holds because in the common case, processes will be able to deliver the required PREPARE and COMMIT messages and decide in the first view, without having to wait for any signatures to be produced or verified. The third feature (the algorithm issues $O(n^2)$ signatures on view-change) holds because, in the worst case, during a view change each process will sign and broadcast a VIEWCHANGE message, thus incurring $O(n)$ signatures in total, and, for each such message, each other process will sign and broadcast a VIEWCHANGEACK message, thus incurring $O(n^2)$ signatures. The fourth feature states that the algorithm issues $O(n)$ required background signatures within a view. These signatures are incurred by *cb-broadcast*ing PREPARE and COMMIT messages. In every view, correct processes broadcast a COMMIT message, thus incurring $n - f = O(n)$ signatures in total.

To the best of our knowledge, no existing algorithm has achieved all these four features simultaneously. The only broadcast-based algorithm which solves consensus with $n = 2f + 1$ processes that we are aware of, that of Correia et al. [50], requires $O(n)$ calls to Reliable Broadcast before any process can decide; this would incur $O(n^2)$ required background signatures when using our Reliable Broadcast implementation—significantly more than our algorithm's $O(n)$ required background signatures.

At this point, the attentive reader might have noticed that our consensus algorithm uses some techniques that bear resemblance to our Reliable Broadcast algorithm in Section 4.5. Namely, the primary of a view *cb-broadcast*s a PREPARE message which is then echoed by the replicas in the form of COMMIT messages. Also, during view change, a replica's VIEWCHANGE message is echoed by other replicas in the form of VIEWCHANGEACK messages. This is reminiscent of the Init-Echo technique used by our Reliable Broadcast algorithm.

Thus, the following question arises: Can we replace each instance of the witnessing technique in our algorithm by a single Reliable Broadcast call and thus obtain a conceptually simpler algorithm, which also satisfies the three above-mentioned properties? Perhaps surprisingly, the resulting algorithm is incorrect. It allows an execution which breaks agreement in the following way: a correct replica p_1 *rb-delivers* some value v from the primary and decides v; sufficiently many other replicas time out waiting for the primary's value and change views

without "knowing about" v; in the next view, the primary *rb-broadcasts* v', which is delivered and decided by some correct replica p_2.

Intuitively, by using a single Reliable Broadcast call instead of multiple Consistent Broadcast calls, some information is not visible to the consensus protocol. Specifically: while it is true that, in order for p_1 to deliver v in the execution above, $n - f$ processes must echo v (and thus they "know about" v), this knowledge is however encapsulated inside the Reliable Broadcast abstraction and not visible to the consensus protocol. Thus, the information cannot be carried over to the view-change, even by correct processes. This intuition provides a strong motivation to use Consistent Broadcast—rather than Reliable Broadcast—as a first-class primitive in the design of Byzantine-resilient agreement algorithms.

5 uBFT: Microsecond-Scale BFT using Disaggregated Memory

Equipped with the knowledge of Chapter 4, namely that it is possible to implement $2f+1$ Byzantine-resilient consensus in the message-and-memory model [3] and avoid expensive cryptography most of the time, we turn our attention to leveraging this knowledge by building a full-fledged Byzantine fault-tolerant (BFT) system for state machine replication (SMR). BFT is essential, as real-life systems occasionally fail in unplanned and unpredictable ways. Apart from simple crashes, failures in distributed systems range from software/configuration bugs [116, 142], to hardware failures [130, 131, 138], to hardly detectable hardware bugs [59, 76, 82, 83] and up to malicious activity [10, 160]. Traditionally, protecting against such different failures required slow and expensive BFT protocols, which pose a prohibitive cost when it comes to microsecond-scale computing.

To this end, this chapter presents uBFT, the first SMR system to achieve microsecond-scale latency in data centers, while using only $2f+1$ replicas to tolerate f Byzantine failures. uBFT relies on a small non-tailored trusted computing base—disaggregated memory—and consumes a practically bounded amount of memory. uBFT is based on a novel abstraction called Consistent Tail Broadcast, which we use to prevent equivocation while bounding memory. We implement uBFT using RDMA-based disaggregated memory and obtain an end-to-end latency of as little as 10 µs. This is at least 50× faster than MinBFT, a state-of-the-art $2f+1$ BFT SMR based on Intel's SGX. We use uBFT to replicate two KV-stores (Memcached and Redis), as well as a financial order matching engine (Liquibook). These applications have low latency (up to 20 µs) and become Byzantine-tolerant with as little as 10 µs more. The price for uBFT is a small amount of reliable disaggregated memory (less than 1 MiB), which in our prototype consists of a small number of memory servers connected through RDMA and replicated for fault tolerance.

5.1 Introduction

The standard way to achieve fault tolerance is state machine replication (SMR). Up until now, a BFT protocol implementing SMR incurs milliseconds of latency [9, 25], requires a

large number of replicas ($3f+1$ to tolerate f failures) [37, 132, 179], consumes unbounded memory [172], and/or relies on a large trusted computing base [20, 93, 172]. These reasons might explain why BFT has had no adoption in data centers.

In this chapter, we propose uBFT, the first BFT SMR system that simultaneously offers four key features: (1) microsecond-scale latency, (2) few replicas ($2f+1$), (3) practically bounded memory, and (4) a small non-tailored trusted computing base. In the common case, uBFT leverages unanimity to replicate requests in as little as 10 µs end-to-end without invoking the trusted computing base or expensive cryptographic primitives. In the slow path—when there are failures or slowness in the network—uBFT uses a novel protocol that combines digital signatures with judicious use of a trusted computing base. The trusted computing base in uBFT is non-tailored and small: rather than trusted enclaves with arbitrary logic such as Intel's SGX [51] or trusted hypervisors [182]—which have large attack surfaces due to their complexity [52, 68]—uBFT relies solely on disaggregated memory, a technology increasingly present in data centers due to the availability of RDMA [166] today and CXL [47] in a few years. The key mechanism from disaggregated memory we leverage in uBFT are single-writer regions (regions of memory that can be written by one designated host and can be read by others), implemented in hardware through access permissions.

Providing the above four features is challenging for BFT protocols. To get microsecond-scale latency, BFT protocols need to avoid expensive public-key cryptography and reduce communication rounds in the common path—and doing so has typically required increasing rather than decreasing the number of replicas [1, 102, 128]. Meanwhile, decreasing the number of replicas has usually required unbounded memory, sophisticated, or tailored trusted computing bases such as append-only-memory [44], SGX [20], TrInc [120], or reliable hypervisors [182]. Limiting the amount of memory is a significant challenge in the design of uBFT. The standard technique—used also in Chapter 4—to handle Byzantine behavior in systems with $2f+1$ replicas requires storing all messages received, leading to long message histories [172], which consume unbounded memory. Finally, not tailoring the trusted computing base to our needs requires designing around existing technologies—in our case disaggregated memory—rather than custom hardware.

To respond to these challenges, uBFT introduces a new abstraction called Consistent Tail Broadcast (CTBcast) that we use to prevent equivocation [125], while requiring a practically bounded amount of memory.[1] Equivocation—a major source of problems in a system with Byzantine failures [44]—occurs when a faulty process incorrectly sends different information to different processes, which may cause the state of replicas to diverge. CTBcast prevents equivocation for all messages, but only ensures the delivery of the last t broadcast messages, where t is a parameter that trades memory for latency (we explain how to set it in Section 5.7).

The price for uBFT is a small amount (less than 1 MiB) of reliable disaggregated memory.

[1] The required memory is logarithmic in the number of operations. Everywhere we use a bounded number of bits except for sequence numbers.

uBFT is designed modularly to work with a generic such component; our current prototype implements this component using RDMA and a set of memory nodes that themselves are replicated for fault tolerance. These nodes add to the total number of replicas, but these replicas are tiny and simple: they do not store the state of the application, just a few in-flight coordination messages. Moreover, their functionality is application independent, so they can be shared among many applications, amortizing their cost. The memory nodes that provide the disaggregated memory constitute the trusted computing base in our prototype and are assumed to fail only by crashing. This shrinks the vulnerability of the system compared to currently deployed crash-tolerant SMR systems, in which *all* components can fail only by crashing, effectively making the trusted computing base be the entire data center.

We evaluate uBFT against two state-of-the-art systems. First, we compare it against Mu, the fastest SMR system to our knowledge, but that tolerates only crash failures. Compared to Mu, uBFT increases the end-to-end latency by only 2×, while tolerating Byzantine failures. Second, we compare uBFT against MinBFT, a state-of-the-art $2f+1$ BFT SMR system, and showcase that our system has more than 50× and 2× better latency when operating in its fast and slow path, respectively. We also use uBFT to replicate two low-latency KV-stores (Memcached [85] and Redis [156]), and a financial order matching engine (Liquibook [139]). All these applications have request latencies of less than 20 µs when unreplicated and become Byzantine-resilient with as little as 10 µs more.

In summary, our main contributions are the following:

- The design of uBFT, a BFT system for state machine replication with microsecond-scale latency in the common case, using only $2f+1$ replicas, practically bounded memory, and a small trusted computing base (disaggregated memory).
- A new abstraction against equivocation, Consistent Tail Broadcast (CTBcast), and a protocol for CTBcast that uses a small amount of disaggregated memory and has a signature-less fast path.
- An open-source implementation of uBFT, CTBcast, and reliable shared disaggregated memory using RDMA, available at https://github.com/LPD-EPFL/ubft.
- A thorough evaluation of the performance of uBFT and its applications.

5.2 Background

5.2.1 Non-Equivocation

Byzantine processes can equivocate, i.e., they can maliciously say different things to different processes. In SMR, specifically, a Byzantine leader may propose different values to try to cause replicas to diverge, justifying why SMR protocols must ensure non-equivocation.

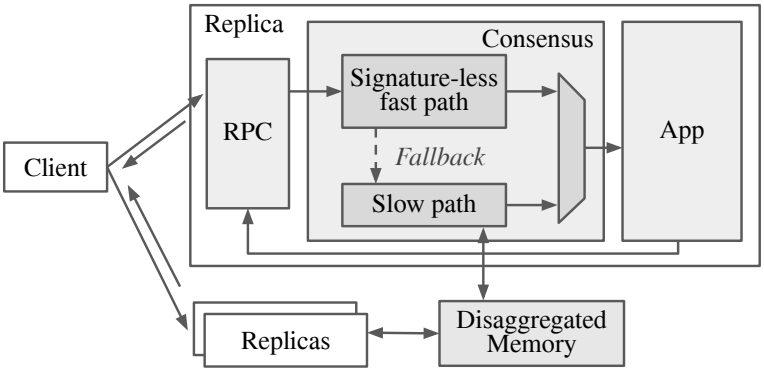

Figure 5.1: Overview of uBFT's architecture.

Under the Byzantine asynchronous model, $3f+1$ replicas are needed to prevent equivocation [113]. However, if equivocation is prevented and transferable authentication is available, Byzantine SMR requires only $2f+1$ replicas [45], the same number as in the crash-stop case. With transferable authentication, a process that verifies a proof about the origin of a message can transfer the proof to other processes and be assured they can also verify it. For example, digital signatures provide transferable authentication, while arrays of Message Authentication Codes (MACs) do not [7].

Preventing equivocation using up to $2f+1$ replicas requires a compromise [45], i.e., a hybrid model where part of the system—called the *trusted computing base*—fails only by crashing. Ideally, this base is as small as possible, since a small and simple base is less likely to be susceptible to failures (e.g., vulnerabilities, bugs).

5.2.2 Disaggregated Memory

Disaggregated memory is an emerging data center technology that separates compute from memory, by providing a shared memory pool that compute nodes access over a network. The memory pool has limited compute capabilities, which it uses for management tasks such as connection handling. Disaggregated memory improves memory utilization, separates the scaling of compute and memory, and achieves better availability due to the separation of fault domains [174].

Disaggregated memory can be provided by different technologies. The emerging CXL standard will support disaggregated memory in the future [74], while today disaggregated memory is available via Remote Direct Memory Access (RDMA) [166] on InfiniBand [87] or RoCE [19]. RDMA is a networking technology that allows a process to read or write the memory of another machine without involving the CPU of the latter. Combined with kernel-bypass, RDMA enables sub-microsecond communication and stringent tail latency. Access rights to RDMA-exposed memory can be set individually for each accessor.

5.2.3 Model

We consider a system with $2f+1$ compute nodes and single-writer multiple-reader disaggregated memory. Up to f compute nodes are Byzantine and may thus fail arbitrarily. We assume network connections are authenticated and tamper-proof (processes know who they get messages from and messages cannot be altered) and eventually available (network partitions are intermittent). We also assume that the disaggregated memory is trusted: it may fail only by crashing. The disaggregated memory is divided into chunks, where each chunk is readable by all compute nodes and writable by a designated compute node. We assume the existence of public-key cryptography: processes can sign messages using their private key and verify unforgeable signatures using the pre-published public keys of all processes. We further assume eventual synchrony: network and processing delays are unbounded until an unknown *Global Stabilization Time* (GST) after which delays are bounded by a known δ. Lastly, our system assumes bounded clock drift for safety, i.e., the clocks of correct processes drift from each other with a bounded rate. These assumptions are common for distributed systems in data centers [6, 37, 65, 119, 177].

In our prototype, we do *not* assume that we are given a reliable disaggregated memory [115, 181], but rather show how to implement a reliable disaggregated memory using RDMA.[2] To do so, we assume $2f_m+1$ memory nodes out of which f_m can fail. Memory nodes are part of the trusted computing base: they are not Byzantine and may fail by crashing only. Memory nodes are simple: they just provide read and write functionality with access control. Their size and functionality do not depend on the application being replicated, and they can be shared among many applications.

5.3 Design

5.3.1 Overview

uBFT follows the design of PBFT [37], a seminal paper that describes how to build practical BFT SMR systems. Figure 5.1 depicts the architecture of uBFT. On the left, a client sends requests to replicas on the right and waits for responses from a majority of them. The replicas go through two stages. First, they totally order client requests using a leader-based BFT consensus protocol. Second, they execute the ordered requests on their local instance of the replicated application before forwarding the outcome of the execution to the client. To achieve microsecond-scale latency, the consensus engine uses a fast/slow path approach. As long as the system is synchronous and all replicas collaborate, the fast path orders requests without signatures. If the fast path does not make progress, uBFT's consensus switches to the slow path, which makes progress with a mere majority of processes using signatures and disaggregated memory.

[2] uBFT's clean encapsulation of disaggregated memory allows future replacement of its RDMA implementation with CXL-powered memory.

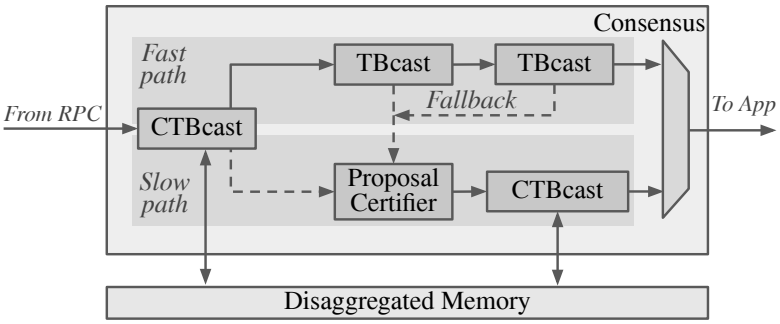

Figure 5.2: Overview of uBFT's consensus engine.

uBFT significantly differs from PBFT in the way it prevents equivocation. PBFT, being a $3f+1$ fault tolerant protocol, relies on intersecting quorums to ensure that malicious replicas do not make the state of honest replicas diverge. By contrast, uBFT operates with $2f+1$ replicas, and therefore cannot rely on the same mechanism as PBFT; instead, it relies on trusted disaggregated memory, which is encapsulated within a new primitive called *Consistent Tail Broadcast* (CTBcast).

CTBcast is a variant of Consistent Broadcast (Section 4.3.1). Consistent Broadcast prevents equivocation by ordering all messages broadcast by a given process. With it, a Byzantine leader is constrained from sending different request orderings to different followers. Our *tail* variant is a relaxation that requires correct processes to deliver only the last t messages sent by a correct broadcaster, while preserving non-equivocation for all messages. This relaxation is essential to practically bound the memory use. Importantly, the implementation of CTBcast has a signatureless fast-path to achieve the latency requirements of uBFT.

Figure 5.2 depicts uBFT's consensus component with its fast/slow path design. After receiving a request from RPC, the leader proposes its ordering via a round of CTBcast. The rest of consensus tries to turn this ordering into a globally accepted one (i.e., stable across leaders). Depending on the synchrony of the system and the number of faulty replicas, this round of CTBcast might execute the fast or slow path. In the former case, consensus continues with its fast path and executes two rounds of Tail Broadcast (TBcast), a form of best-effort broadcast designed for finite memory (§5.4.1). Importantly, none of these three broadcasts involve signatures nor disaggregated memory. If liveness is lost during the fast path of consensus, uBFT activates the slow path, shown by the dashed arrows, which executes a certification round and another round of CTBcast. The slow path is also executed if the fast path of the initial CTBcast fails. Differently from the three rounds of the fast path, the three rounds of the slow path all require signatures, and CTBcast invocations additionally require disaggregated memory.

5.3.2 Challenges

uBFT addresses the following challenges:

2f+1 replicas and finite memory. Previous theoretical work [4] proposed to prevent equivocation with fewer than $3f+1$ replicas by building Consistent Broadcast on top of shared registers. However, this abstraction requires replicas to use infinite memory in order to store and deliver *all* broadcast messages, which is not implementable in practice. We work around this memory issue by designing CTBcast, a weaker form of Consistent Broadcast where replicas are allowed to skip the delivery of old messages in order to favor the delivery of newer ones (§5.4).

SMR with CTBcast. The reliance of uBFT on CTBcast brings additional complexity to its consensus algorithm, notably on preventing equivocation *across* messages. Typically, protocols rely on the entire history of messages to prevent equivocation. Yet, CTBcast only guarantees the delivery of the tail, which may lead correct replicas to have gaps in their delivery history. uBFT works around this limitation via *CTBcast summaries*, which allow a replica to make progress in spite of gaps (§5.5).

Microsecond-scale operation. Systems that operate at the microsecond scale should avoid signatures on their critical path. Yet, Chapter 4 shows, Consistent Broadcast cannot completely remove signatures. Moreover, recycling memory requires the generation of proofs which also involve signatures. uBFT addresses this challenge by avoiding expensive cryptography in the fast path of CTBcast and relegating the few bookkeeping signatures to a background task (§5.5.4).

Resilient disaggregated memory. uBFT relies on RDMA to implement disaggregated memory. However, raw memory exposed over RDMA is not enough to implement our SMR protocol. Indeed, RDMA-exposed memory does not tolerate failures, and data accesses can be inconsistent, since RDMA provides only 8-byte atomicity. uBFT addresses these limitations of RDMA using efficient, yet Byzantine fault tolerant, algorithms (§5.6).

5.4 Consistent Tail Broadcast

Consistent Tail Broadcast (CTBcast) is a novel variant of Consistent Broadcast (CBcast) that uBFT uses to prevent equivocation. Briefly, CTBcast resembles CBcast, except that it allows processes not to deliver outdated messages. In this way, CTBcast avoids maintaining the full history of messages, to bound memory use.

5.4.1 Definition

CTBcast is defined in terms of two primitives, *broadcast(k,m)* and *deliver(k,m,p)*, where k is a numeric identifier, m is a message, and p is a process. When p invokes *broadcast(k,m)*, we say that p broadcasts (k, m), i.e., it broadcasts message m with identifier k. A correct broadcaster

increments k sequentially at every broadcast, starting with $k = 1$. Similarly, when a process q invokes *deliver(k,m,p)*, we say that q delivers (k, m) from p.

In simple terms, CTBcast is a multi-shot abstraction that prevents correct processes from delivering different messages from a given broadcaster p for the same identifier k. CTBcast is parameterized by a tail t, which specifies which messages are guaranteed to be delivered. Informally, in CTBcast, a correct process q is only required to deliver the last t messages broadcast by a correct process p, while the delivery of previous messages is best-effort.

CTBcast has the following properties:

Tail-validity If a correct process p broadcasts (k, m) and never broadcasts a message (k', m') with $k' \geq k + t$, then all correct processes eventually deliver (k, m).

Agreement If p and q are correct processes, p delivers (k, m) from r, and q delivers (k, m') from r, then $m = m'$.

Integrity If a correct process delivers (k, m) from p and p is correct, p must have broadcast (k, m).

No duplication No correct process delivers $(k, *)$ from p twice.

The difference between CTBcast and CBcast lies in their validity property. Tail-validity implies that a correct process is only obliged to deliver a message m from p if m is among the last t messages broadcast by p. When $t{=}\infty$, tail-validity reduces to CBcast's validity.

The infinite tail of CBcast is what prevents it from recycling memory. Indeed, given that the broadcaster cannot distinguish between network asynchrony and receiver failures [56], it is required to keep re-transmitting all messages until they are explicitly acknowledged. Thus, in CBcast, the broadcaster can garbage collect messages only after they have been acknowledged by all receivers. As a result, once a single process fails, memory cannot be recycled and any correct implementation of CBcast must block after running out of memory. By not enforcing the delivery of *old* messages, CTBcast's tail-validity lets processes recycle the memory dedicated to these messages. This is why CTBcast requires only finite memory while CBcast does not. In Section 5.5, we show that despite its weaker semantics, CTBcast is sufficient for solving consensus.

5.4.2 Algorithm

Algorithm 5.1 implements CTBcast using finite memory with a fast/slow path approach that avoids signatures and disaggregated memory in the common case. For pedagogical reasons, we assume that a designated process is the broadcaster while the others are receivers. Each receiver owns an array of t Single-Writer Multiple-Reader (SWMR) regular registers. Each register is only writable by its owner, but is readable by all processes. The regularity of the

registers forces READs that execute concurrently to a WRITE to return either the value being written or the previous one. Moreover, each process uses a Tail Broadcast (TBcast) primitive which ensures the delivery by correct processes of the last $2t$ messages broadcast through it, but does not prevent equivocation. Formally, TBcast has all properties of CTBcast except agreement.

Implementing TBcast using finite memory is simple. The broadcaster buffers its last $2t$ messages and retransmits them until it receives acknowledgements from all receivers. To broadcast a new message when the buffer is full, the broadcaster makes room for it by evicting the oldest buffered message.

As mentioned, Algorithm 5.1 has a low-latency fast path that avoids signatures and disaggregated memory. It also incorporates a fall-back slow path for liveness. For presentation simplicity, it triggers the slow path in parallel to the fast path (lines 3 and 4), but in reality, uBFT triggers it when replicas fail to decide on new client requests after some configurable duration. In addition to the shared SWMR registers, receivers use three finite-size local arrays for bookkeeping (lines 8-10).

In the fast path, the broadcaster first TBcast-broadcasts its message alongside its identifier within a `LOCK` message (line 3). When receivers deliver this message (line 12), they commit not to deliver any other message for the given identifier, and tell other receivers about their commitment by broadcasting a `LOCKED` message (line 16). Receivers use `locks` (lines 13-15) to avoid committing to different messages for the same identifier. Importantly, receivers store only up to t commitments in this array, by evicting earlier commitments that alias to the same index $k\%t$ (line 15). When receivers learn that everyone committed to the same message (lines 18-22), they know that no correct replica will deliver a different message and thus deliver it (line 23).

In the slow path, the broadcaster additionally TBcast-broadcasts a signed version of its message (line 4). After TBcast-delivering a signed message (line 25), receivers verify its signature (line 26). Then, they check that they have not committed to a different message for the same k (line 28), and ensure that they will not do so in the future (line 29). Subsequently, they copy the signed message to their SWMR register associated with the message identifier k (line 30), before reading the associated SWMRs owned by other receivers (line 31). Receivers ignore messages with invalid signatures (line 32) and abort delivery if they detect a different message for the same identifier (line 33). In case receivers detect another message with a higher identifier that is associated with the same SWMR registers (line 35), they drop their own message as it no longer belongs to the tail. Otherwise, they deliver it (line 37).

The correctness of the slow path of Algorithm 5.1 hinges on that all correct processes will find the message copied by the *fastest* correct replica when reading the registers. Thus, they can deliver no message other than the first copied, hence preserving agreement. The fast and slow paths are linked together via the `locks` array (lines 15 and 29), which ensures that whichever path executes first forces the value of the message for the other path. Note that when the

Algorithm 5.1: Consistent Tail Broadcast

```
# at the broadcaster
def broadcast(k, m):
  TBcast-broadcast ⟨LOCK, k, m⟩
  TBcast-broadcast ⟨SIGNED, k, m, sign((k, m))⟩

# at receivers:
SWMR[me] = [(−1, ⊥, ⊥), ... ] # array of t slots
delivered = [−1, ... ] # array of t slots
locks = [(−1, ⊥), ... ] # array of t slots
locked = [[(−1, ⊥), ... ], ... ] # array of t*n slots

upon TBcast-deliver ⟨LOCK, k, m⟩ from p:
  k', _ = locks[k%t]
  if k > k':
    locks[k%t] = (k, m)
    TBcast-broadcast ⟨LOCKED, k, m⟩

upon TBcast-deliver ⟨LOCKED, k, m⟩ from q:
  k', _ = locked[q][k%t]
  if k > k':
    locked[q][k%t] = (k, m)
    if locked[r_0][k%t] == ... == locked[r_{n−1}][k%t]:
        deliver_once(k, m)

upon TBcast-deliver ⟨SIGNED, k, m, sig⟩ from p:
  if valid(sig, (k, m), p):
    k', m' = locks[k%t]
    if k > k' or k == k' and m == m':
      locks[k%t] = (k, m)
      SWMR[me][k%t].write((k, sig, m))
      for each (k', s', m') in SWMR[*][k%t]:
        if valid(s', (k', m'), p):
          if k' == k and m' != m:
            return # Byzantine broadcaster
          if k' > k and k' ≡ k (mod t):
            return # out of tail
      deliver_once(k, m)

def deliver_once(k, m):
  if k > delivered[k%t]:
    delivered[k%t] = k
    trigger deliver(k, m)
```

broadcaster is Byzantine, correct processes are allowed not to deliver. A more detailed proof of correctness is given in Appendix C.1.

5.5 State Machine Replication

Like all leader-based BFT consensus protocols, uBFT's protocol has the same high-level layout as PBFT [37]: it shares naming conventions and splits the protocol in similar phases. However, uBFT has different goals ($2f+1$ fault tolerance, finite memory, microsecond latency),

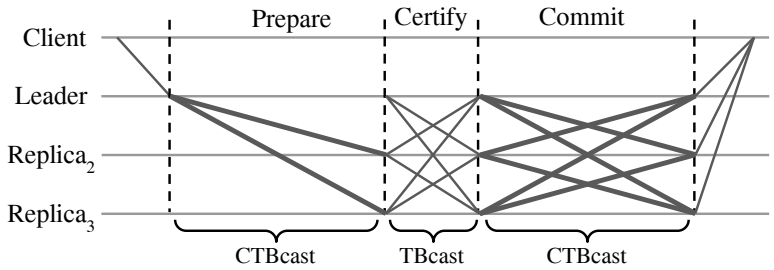

Figure 5.3: Communication pattern of uBFT's slow path. Bold lines represent messages sent over CTBcast. Thinner lines represent direct messages.

different assumptions (disaggregated memory), and a different non-equivocation mechanism (Consistent Tail Broadcast (CTBcast)). As a result, uBFT's protocol is structurally different from PBFT, as we describe in this section.

We start by giving an overview of our consensus protocol. Then, we analyse its slow path and explain how it deals with finite memory and view changes. Finally, we describe how its fast path leverages unanimous and timely collaboration of replicas to achieve microsecond-scale latency. For presentation clarity, this section gives only an informal description of our protocol. Its pseudocode is given in Appendix C.2 alongside detailed arguments of its correctness.

5.5.1 Basic Protocol

From a high-level point of view, the slow path of our consensus protocol—shown in Figure 5.3—has three phases: *Prepare*, *Certify* and *Commit*. After the leader receives a signed request from a client, it *proposes* it by broadcasting a `Prepare` message via CTBcast. When replicas (including the broadcaster) deliver this message, they proceed to its certification. Each replica signs the proposal and TBcast-broadcasts its signature in a `CERTIFY` message. Then, it waits to aggregate $f+1$ signatures on the proposal, which constitute an unforgeable proof that the proposal was emitted by the leader. Moving on to the Commit phase, the replicas broadcast via CTBcast a `COMMIT` message containing the aforementioned proof. Finally, after delivering $f+1$ `COMMIT` messages, replicas apply the client's request to their local state machine and reply to the client. Note that we use TBcast instead of the more expensive CTBcast when broadcasting `CERTIFY` messages: equivocation of `CERTIFY` messages does not harm correctness as all certificates involve at least one correct replica, and all correct replicas certify the same proposal due to the CTBcast in the Prepare phase.

So far, the described protocol replicates a single client request. Similarly to PBFT, uBFT uses a sliding window to run its consensus protocol on a series of slots. As the leader receives multiple requests, it proposes each one in a different slot, handling many slots in parallel. uBFT also uses PBFT's application checkpoints to throttle the number of concurrent request proposals. This mechanism limits the impact of a Byzantine leader, and bounds the number of relevant messages at any point in time. An application checkpoint is signed by $f+1$ replicas

and includes (1) the state of the application after applying the first i replicated requests to it, and (2) an implicit authorization to work on slots $[i+1, i+window]$.

5.5.2 Non-Equivocation at the Consensus Level

An important aspect of uBFT's consensus protocol is how it prevents the leader from sending conflicting proposals. CTBcast only partially solves this issue: it prevents a Byzantine leader from sending conflicting proposals in a single message, but it does not prevent it from sending conflicting proposals across multiple messages. Conceptually, a process can ensure that another process has not equivocated if the former knows the entire history of messages broadcast by the latter. For this reason, processes in our protocol interpret messages of other processes in FIFO order. However, CTBcast does not guarantee the delivery of all broadcast messages due to its tail-validity property, which might prevent processes from delivering *all* the messages from a correct one in FIFO order. We solve this issue by pairing CTBcast with *CTBcast summaries*.

A CTBcast summary is an unforgeable synopsis of what has been broadcast by a process p via CTBcast up to a given CTBcast identifier. Summaries constitute certificates that are signed by $f+1$ replicas, which have witnessed the messages broadcast by p and assert that p has not equivocated at the consensus level. When a process receives a summary about p up to a certain CTBcast identifier i, it is able to continue handling p's new messages above i in FIFO order. Essentially, CTBcast summaries restore FIFO delivery that may have been broken by the tail-validity of CTBcast.

CTBcast summaries are generated interactively. Every t consecutive CTBcast messages (t is CTBcast's tail parameter) that a replica r delivers from p, r participates in creating a summary about the state of p. Replica p knows how many messages it has broadcast and blocks waiting for a summary of its state every t messages. Using its previous summary, p can bring correct processes that missed some of its messages up to speed, and help them deliver the last t messages it broadcast so far. Even if f Byzantine replicas fail to help building p's next summary, all correct replicas would eventually collaborate to generate it, thus ensuring the liveness of the summaries and allowing p to keep broadcasting.

It is important to ensure that CTBcast summaries have finite size. To this end, a process keeps only a limited number of messages that it CTBcast-delivers from others, by maintaining a window of consensus slots for every other broadcasting process. If a process p CTBcast-delivers a message from another process q out of the window it maintains for q, p considers q as being Byzantine. uBFT requires processes to CTBcast-broadcast application checkpoints in order to slide their window on the receivers' side. When a receiver slides a broadcaster's window forward, it drops messages referring to slots that fall out of it, since the consensus slots they refer to have been checkpointed. This way, the entire relevant state of a broadcaster is a combination of a consensus window range and the collection of CTBcast messages that fall into it.

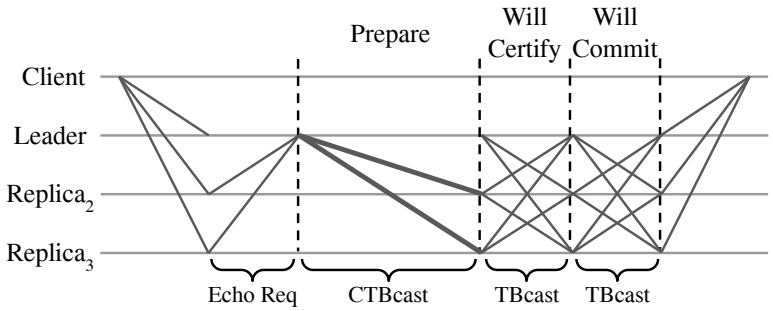

Figure 5.4: Communication pattern of uBFT's fast path. Bold lines represent messages sent over CTBcast. Thinner lines represent direct messages.

5.5.3 View Change

To tolerate faulty leaders, uBFT follows PBFT's *view change* approach. Briefly, every view has a dedicated leader determined in a round-robin manner. If a replica does not make progress or believes that the leader is censoring client requests, it moves to the next view by CTBcast-broadcasting a `SEAL_VIEW` message. The leader of the new view first transfers any potentially applied requests to the new view by CTBcast-broadcasting a `NEW_VIEW` message before proposing new requests, as detailed below.

The `NEW_VIEW` message contains the latest application checkpoint, as well as all `COMMIT` messages about the open slots following this checkpoint from $f+1$ replicas. More precisely, when a replica p receives a `SEAL_VIEW` message from another replica q, p proceeds with generating a certificate share about q's state. In fact, this state includes q's latest application checkpoint, as well as the latest `COMMIT` messages sent by q for each one of q's open slots. The content of the `NEW_VIEW` message consists of $f+1$ matching certificate shares about $f+1$ different replicas. The broadcast `NEW_VIEW` message constrains the values that the new leader can propose for open slots. That is, for every open consensus slot, the new leader is required to propose the value of the `COMMIT` message with the highest view number (if any).

The view change mechanism prevents a leader from omitting requests that were decided in previous views. Intuitively, if a correct replica decided on a request in some view, it must have received $f+1$ matching `COMMIT` messages. As the leader collects certificates about $f+1$ replicas, it must necessarily include at least one certificate with a `COMMIT` message for all decided requests.

5.5.4 Fast Path

To operate at the microsecond scale, uBFT incorporates a fast path—shown in Figure 5.4—that moves signatures out of the critical path in times of synchrony and unanimous collaboration. Similar to the slow path, the fast path has three phases: Prepare, which is common with the slow path but executes the fast path of CTBcast, followed by *WillCertify* and *WillCommit*. The

last two phases replace the Certify and Commit phases of the slow path with inexpensive rounds of Tail Broadcast.

The operation of the fast path is simple. For a given consensus slot, after the end of the Prepare phase, replicas broadcast a `WILL_CERTIFY` message and wait to receive the same message from all others. Once received, they proceed with broadcasting a `WILL_COMMIT` message and again wait for unanimity before deciding on the proposed value. Both messages contain solely the view number and the consensus slot. These messages are essentially promises that their broadcaster will run the slow path before CTBcast-broadcasting its next `SEAL_VIEW` message. By broadcasting `WILL_CERTIFY`, a replica promises to participate in certifying the `PREPARE` message. With `WILL_COMMIT`, it promises to CTBcast-broadcast the resulting certificate within a `COMMIT` message.

The safety of this scheme is intuitive. If a replica receives $2f+1$ `WILL_CERTIFY` messages, it knows that at least $f+1$ correct replicas will certify the `PREPARE` message in the Certify phase. Similarly, receiving $2f+1$ `WILL_COMMIT` messages means that at least $f+1$ correct replicas will send a `COMMIT` message in the Commit phase and thus no other value will be decided for this slot.

The fast path also takes care of finite memory as it does not keep promises forever: it drops the promises that refer to consensus slots included in an application checkpoint. These checkpoints along with the CTBcast summaries make up the required background signatures of uBFT's fast path.

Lastly, the fast path eschews signatures between clients and replicas by having the clients send unsigned requests to all replicas, instead of a signed request to the leader. A replica endorses a `PREPARE` message on a client request only if it also received it from the client. Thus, the fast path contains an additional communication round (denoted *Echo Req*) in which the leader waits for the followers to echo the client request before proposing it. This way, a Byzantine client cannot cause a leader to get stuck (and thus incur a view change) by not sending its requests to all correct replicas.

5.6 Implementation

The design of uBFT relies on disaggregated memory to build the shared registers used by CTBcast, and on a fast networking fabric to achieve its microsecond-level latency. This section explains how we use RDMA [166] to achieve these goals.

5.6.1 Reliable SWMR Regular Registers

The CTBcast component of uBFT requires reliable SWMR regular registers. *Reliable* means that the registers do not fail, i.e., `READ`s and `WRITE`s always complete. *SWMR* means Single-Writer Multiple-Reader: each register has an owner, which is a replica that is allowed to write to it,

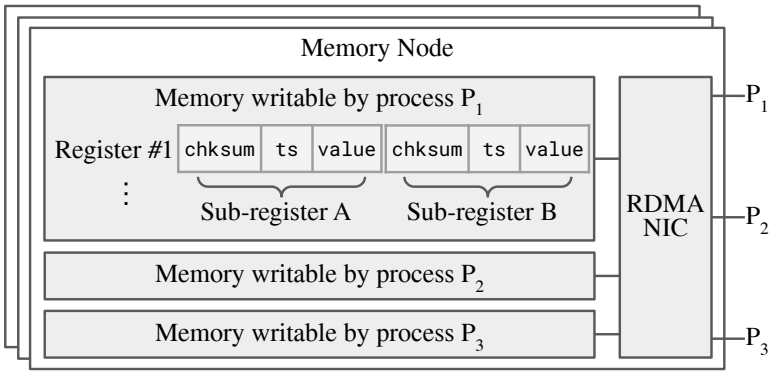

Figure 5.5: Reliable SWMR Regular registers using RDMA.

while all other replicas may only read the register. *Regular* means that, when a READ executes concurrently to a WRITE, the former should return either the value that is being written or the previous one. We explain below how to implement each property using RDMA.

SWMR register. We implement SWMR registers using RDMA permissions. RDMA splits memory into regions, each with different access permissions based on a token. We create a read-write and a read-only region for the same memory range, give the read-write token to the writer of the register, and the read-only token to the other replicas.

Regular register. RDMA-exposed memory is atomic (hence regular), but only at an 8-byte granularity. While RDMA WRITEs have left-to-right ordering, RDMA READs do not. Thus, an RDMA READ concurrent with a WRITE may return partially written data, mixing old and new values. To detect this problem, we use checksums, as in Pilaf [133]. A simple approach, where a reader retries until the checksum is valid, violates liveness as a Byzantine writer can write bogus checksums. To avoid such scenario, we follow an evolved double-buffering strategy, which ensures that a reader is always able to find a complete WRITE or detect the owner of the register as being Byzantine. As depicted in Figure 5.5, each register is made of two sub-registers. Each WRITE to a given register is directed to one of the sub-registers in a round-robin manner. To write a value, the writer prefixes it with a logical timestamp (denoted ts) and a checksum. Importantly, the writer waits for δ (the known communication bound after GST) between two WRITES to the same register. To perform a READ, the reader reads both sub-registers at once and, out of the values with a valid checksum, returns the one with the highest timestamp. [3] If both checksums are invalid and the READ took less than δ, the writer is Byzantine and a default value is returned. Otherwise, the READ is retried. The writer is also deemed Byzantine if both

[3]The hardware is allowed to reorder RDMA READs following RDMA WRITEs when issued to different Queue Pairs (QPs) [11]. To ensure regularity, i.e., that subsequent register READs see the RDMA-written value, a register WRITE only returns after the PCIe WRITE transaction reaches the last-level cache (L3). We do so by issuing an RDMA READ after the RDMA WRITE to the same QP—which acts as a PCIe fence [75]—and only considering that the register WRITE completes when the RDMA READ completes.

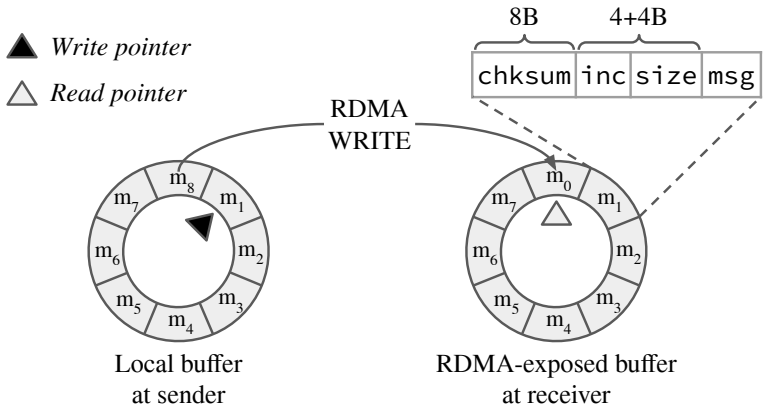

Figure 5.6: uBFT's RDMA-powered message-passing primitive.

sub-registers have the same timestamp. This scheme works in an eventually synchronous system with bounded-drift clocks [65]: when the writer and a reader are correct, a `READ` does not overlap with two `WRITE`s, so one of the sub-registers will have a valid checksum, and ordering based on timestamps ensures regular register semantics.

Reliable register. We replicate each register to $2f_m+1$ memory nodes, where each memory node exposes its memory over RDMA (Figure 5.5). Here, f_m is the maximum number of memory nodes that may crash. While memory nodes add to the total number of replicas in the system, these nodes do not replicate the application, and they can be shared among many replicated applications, as each application takes a small amount of space in the memory nodes (§5.7.6). Our register replication scheme is straightforward. `WRITE`s are issued to all memory nodes in parallel and return after having completed at f_m+1 of them. `READ`s are also issued to all nodes in parallel, wait for f_m+1 of them to complete, and return the value of the regular register with the highest timestamp. This scheme tolerates Byzantines replicas, as `READ`s and `WRITE`s always complete at a majority, and preserves regular semantics. Indeed, when a `READ` is not concurrent to any `WRITE`, intersecting quorums ensure that the last written value is returned. In case of concurrency, the `READ` intersects with the concurrent `WRITE` in some register and with the last completed `WRITE` in some other register. Thus, it will return a value no older than the value of the last completed `WRITE`, ensuring regularity.

5.6.2 A Fast Message-Passing Primitive

To achieve microsecond-scale communication, uBFT implements a fast one-way messaging primitive between a sender and a receiver, where the receiver is required to deliver only the last t messages of the sender, similarly to CTBcast. This primitive allows an implementation without receiver acknowledgements, which we found to be important for microsecond-scale performance.

Figure 5.6 depicts the implementation of this primitive over RDMA. The receiver has a circular buffer exposed over RDMA; the buffer is divided into t slots of equal size large enough for the largest message. Briefly, the sender RDMA-writes messages to the receiver's buffer, while the latter scans its local buffer for new messages. There are no acknowledgments: the sender overwrites old messages with newer ones, even if they were not yet received.

We now explain this implementation in detail. The sender allocates a mirror image of the receiver's buffer in its local memory, and maintains locally a *write pointer* to the slot for its next message. Each slot has a header composed of a checksum, an incarnation number (the number of times it was written), and a message size. To send a new message, the sender writes it to the slot pointed by the writer pointer and fills its header. Then, it issues an RDMA `WRITE` to the corresponding slot in the receiver's memory, and marks the slot as unavailable until it is notified of the completion of the `WRITE` by the RDMA NIC. Finally, the sender advances its write pointer. If the pointed slot is unavailable, the new message is queued in a second (not depicted) circular buffer. This buffer acts as a staging area: it forwards its messages for transmission when slots become available, and evicts the oldest queued message to accommodate a newer one.

The receiver maintains a *read pointer* to the slot where it will read the next message. The receiver polls this slot for a particular incarnation number, which identifies the next message it expects to find. Once this incarnation number is seen, the receiver copies the entire message to a private buffer in order to avoid interfering `WRITE`s on the same slot. Then, the receiver checks the incarnation number again in the copied message. If the incarnation number has not changed, the sender verifies the checksum before delivering the message (rescheduling the polling if the checksum is invalid). If the receiver finds a higher incarnation number than expected, it concludes that some older messages may exist in other slots of the buffer that will have to be delivered first. So, the receiver aborts the delivery and advances its pointer to the oldest undelivered message. With this strategy, the receiver guarantees FIFO delivery of the last t messages.

This scheme has two benefits: it uses practically bounded memory and avoids acknowledgements. The latter—even when batched—increase the application's tail latency as scheduling an acknowledgement alone takes ≈300 ns [91], which is time lost handling incoming events. Instead, by the End-to-End Principle [155], acknowledgements are piggybacked in SMR-level messages.

5.7 Evaluation

We evaluate the various performance traits of uBFT and verify its suitability as a BFT SMR system for microsecond applications. We aim to answer the following:

- How much latency does uBFT induce on the replicated applications (§5.7.1)?

Table 5.1: Hardware details of machines.

CPU	2x Intel Xeon Gold 6244 CPU @ 3.60 GHz (8 cores/16 threads per socket)
NIC	Mellanox ConnectX-6 MT28908
Switch	Mellanox MSB7700 EDR 100 Gbps
OS/Kernel	Ubuntu 20.04.2 / `5.4.0-74-generic`
RDMA Driver	Mellanox OFED `5.3-1.0.0.1`

- How does the replication latency of uBFT compare to other SMR systems (§5.7.2)?
- How do the internal components of uBFT contribute to its end-to-end latency (§5.7.3)?
- How does our implementation of CTBcast perform in comparison to SGX-based non-equivocation mechanisms (§5.7.4)?
- How does the tail parameter of CTBcast impact uBFT's tail-latency (§5.7.5)?
- What is the memory consumption of uBFT (§5.7.6)?

We evaluate uBFT in a 4-node cluster, the details of which are given in Table 5.1. The dual-socket machines have an RDMA NIC attached to the first socket. Our experiments execute on cores of the first socket using local NUMA memory.

Our implementation measures time using the `clock_gettime` function with the `CLOCK_MONOTONIC` parameter. The function uses the TSC clock source of Linux for efficient and accurate timestamping [151].

In all experiments we deploy 1 client and 3 replicas, and take at least 10,000 measurements. Additionally, we set the consensus window to 256 requests and—unless stated otherwise—the tail parameter of CTBcast to 128 messages.

Applications. We integrate uBFT with MemCached [91], Redis [156] and Liquibook [139]. MemCached and Redis are non-replicated high-performance KV-stores. Liquibook is a data structure that implements a financial order matching engine. We also integrate all the aforementioned applications with Mu [5], the SMR system that has the lowest replication latency (to our knowledge) but tolerates only crash faults. In all applications, the client sends messages using uBFT's RPC mechanism. Additionally, using a no-op application, we compare uBFT against MinBFT [172], a state-of-the-art $2f+1$ BFT SMR system with a publicly available implementation which uses Intel's SGX [51]. SGX provides a secure CPU enclave for executing arbitrary code, thus offering a general-purpose trusted computing base.

Implementation Effort. We implemented uBFT by extending the framework we developed for uKharon. Our prototype spans 11 750 lines of C++, out of which 2 966 are dedicated to

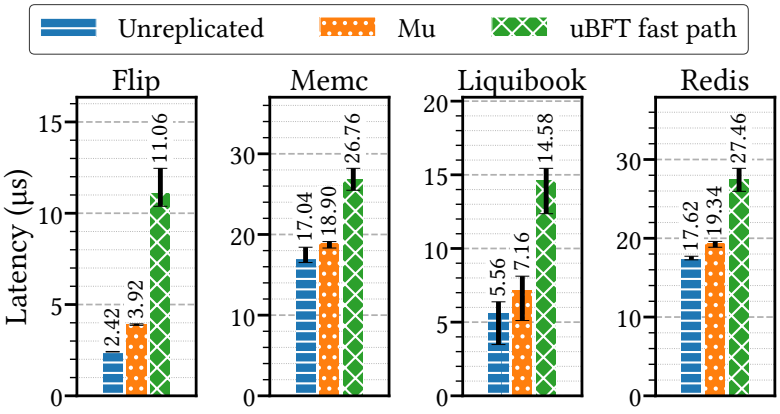

Figure 5.7: End-to-end latency of different applications when either not replicated or replicated via Mu and uBFT's fast path. The printed values are the 90th %-iles. The whiskers show the 50th and 95th %-iles.

CTBcast. The prototype includes all features on the critical path of a complete implementation: the only major unimplemented features are application and replica state transfers. We use Dalek's implementation of EdDSA over Curve25519 [123] for public-key cryptography, BLAKE3 [141] for HMACs and xxHash [46] for checksums.

5.7.1 End-to-End Application Latency

Figure 5.7 explores the replication overhead that uBFT induces to end applications. We compare the latency of its fast path against the unreplicated latency, and the latency when replicating with Mu. We study four applications: Flip, a toy application that reverses its input, as well as Memcached, Redis, and Liquibook.

The KV-stores use 16 B keys and 32 B values. Our workload is 30% `GET`s, out of which 80% succeed and return a non-empty value. Liquibook's requests are 32 B. Its responses range from 32 B to 288 B, depending on how many orders match. 50% of the orders are `BUY` and 50% `SELL`. Finally, Flip's requests and responses are 32 B long.

From the figure, observe that uBFT is consistently slower than Mu by approximately 7.5 µs at the 90th percentile. uBFT's additional induced end-to-end latency is most significant for ultra fast applications, such as Flip, for which our system is 3 times slower than Mu. As the application's unreplicated latency increases, uBFT's latency overhead diminishes. For Liquibook, it is 2× slower than Mu and for the KV-stores, it is only 1.5× slower. At the same time, uBFT's replication increases the latency variance (i.e., the difference between the 50th and 95th percentiles) compared to Mu, which is a consequence of the complexity of our system. More precisely, uBFT's RPC requires one more round of communication than Mu's to ensure that all correct replicas have received the client request. Additionally, uBFT's replication scheme requires 4 rounds of broadcast before replying to the client instead of a single majority

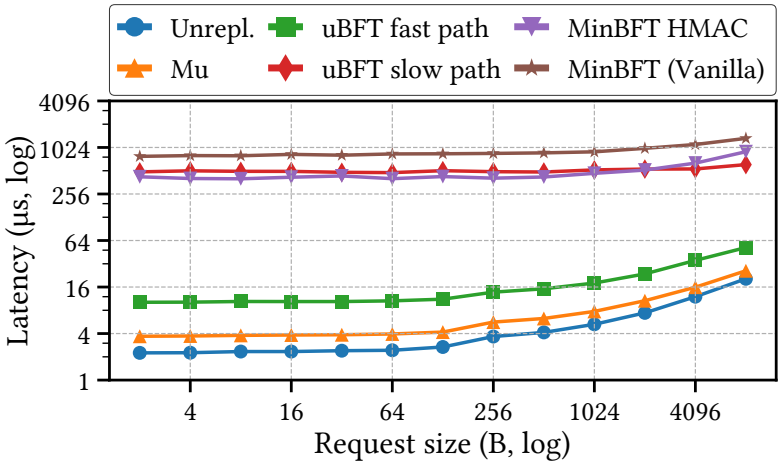

Figure 5.8: Median end-to-end latency for different request sizes of an unreplicated no-op application, as well as its latency when replicated with Mu, uBFT and MinBFT.

`WRITE` in Mu. Overall, uBFT's fast path has 4 additional rounds of communication compared to Mu, hence the higher tail latency. Nevertheless, uBFT does not worsen the variance by more than 3 µs.

In a nutshell, with uBFT, assuming a timely network and the absence of failures, an unreplicated application that operates at the microsecond-scale envelope requires at most an extra 10 µs to become Byzantine fault tolerant.

5.7.2 End-to-End Replication Latency

We now explore the impact of the size of requests on the end-to-end latency of uBFT. Figure 5.8 shows the median client-side latency for various request sizes to a no-op application with different replication schemes. The service replies with responses matching the size of requests.

As expected, the lowest latency is achieved without any replication (denoted *Unrepl.*). Here, the end-to-end latency ranges from 2.2 µs to 20 µs, which is attributed to communicating with the server side using our RPC. The other lines show the service replicated with: Mu, our BFT SMR solution, and MinBFT, a $2f+1$ BFT alternative.

Mu's replication increases the end-to-end latency by up to 64% for small requests and by at most 26% for 8 KiB ones. In the absence of failures, Mu's leader replicates the requests it receives by just RDMA-writing them to its followers. uBFT's fast path exhibits higher latency than Mu due to its 4 additional rounds of communication, yet it only increases the overhead compared to the latter by at most 175%, even though it offers Byzantine fault tolerance.

One could expect that uBFT's fast path would come at the cost of high latency in the slow path, yet this is not the case. We compare our slow path against MinBFT, with the latter operating in two configurations. In its vanilla configuration, MinBFT uses HMACs only between the replicas,

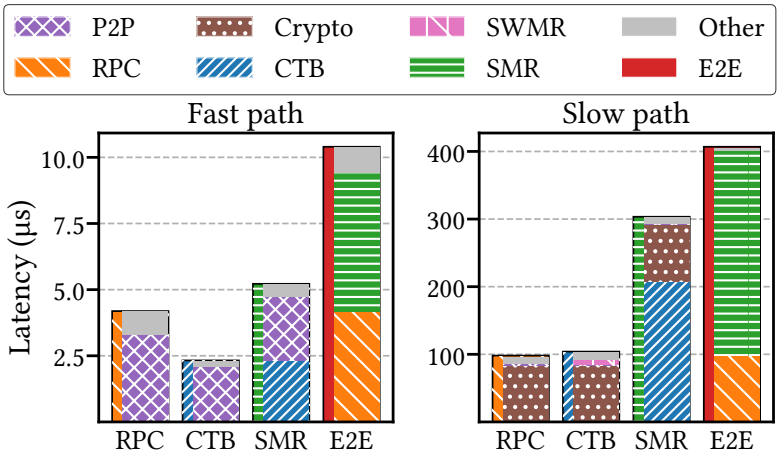

Figure 5.9: Recursive decomposition of the end-to-end latency of uBFT's fast and slow path when replicating Flip with requests of 8 B.

however the clients sign the requests sent to the service using public-key cryptography, leading to a minimum end-to-end latency of 566 µs. We modify MinBFT to also use SGX in the clients, thus replacing their expensive cryptography with HMACs. Since our setup does not offer SGX, we stub it using the latency results of Section 5.7.4. Note that MinBFT is not an RDMA-tailored application and it uses standard TCP in its implementation.

To increase the fairness of our comparison with MinBFT, we use Mellanox's VMA library [167] to replace MinBFT's TCP stack with a kernel-bypass alternative that uses RDMA NICs for increased performance. The end result is that uBFT's slow path is faster than vanilla MinBFT (up to 109%) and at most 24% slower than its purely HMAC-based variant, even though our slow path uses public-key cryptography.

5.7.3 Latency Breakdown

To get a better understanding of uBFT's end-to-end latency, we analyze the latency of its internal components.

Figure 5.9 shows a recursive decomposition of the latency of an 8 B Flip request into its constituents: remote-procedure call, Consistent Tail Broadcast and replication (denoted *RPC*, *CTB* and *SMR* respectively). The rightmost columns of the figure show the client-perceived latency (denoted *E2E*). Each bar has two regions: the narrow left one shows the overall latency of the component, while the wider right one shows its decomposition.

In our decomposition, we identify four primitive sources of latency. First is the time for communication over our message-passing primitive (denoted *P2P*). Next is the time for producing and verifying signatures (denoted *Crypto*), and the time for reading and writing to the disaggregated memory registers (denoted *SMWR*). Both of these are only relevant in uBFT'slow path. The `Crypto` primitive goes beyond pure cryptographic computation, as it

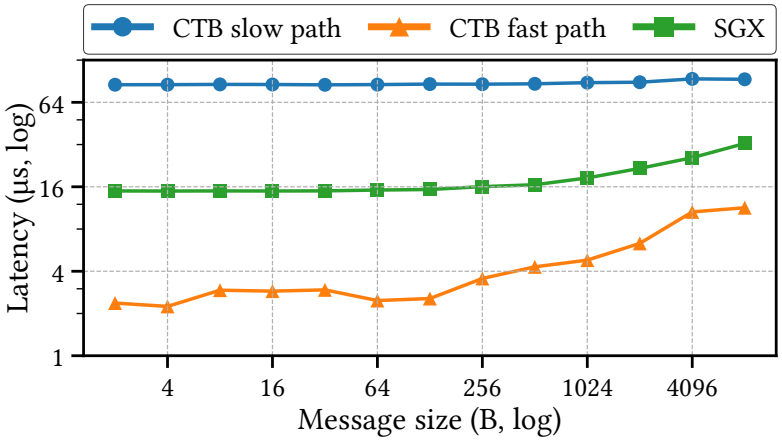

Figure 5.10: Median latency of multiple non-equivocation mechanisms for different message sizes.

includes the synchronization cost of issuing the operation to a thread pool and getting back the result. Lastly, there is the time spent in bridging these basic primitives and connecting the components (denoted *Other*). This last category includes copying buffers, delays between the arrival and processing of asynchronous events, etc.

We can see that, in the fast path, most of the time is spent in communication. Given the small size of the messages, the potential avenues for improving the overall latency is to either reduce the number of communication steps or reduce the latency of the underlying network fabric. In the slow path, public-key cryptography dominates. Given that signatures are unavoidable in our setting, as we established in Chapter 4, achieving better latency requires faster cryptographic primitives. Also, notice that the cost of accessing disaggregated memory—which is part of CTB—is negligible, as it accounts for 14 µs (3.5%) of the end-to-end latency.

5.7.4 Latency of Non-Equivocation

Figure 5.9 demonstrates that CTBcast accounts for a substantial portion of the end-to-end latency of our system's fast and slow paths. By using disaggregated memory, our implementation of CTBcast has an advantage over solutions that use CPU enclaves, since disaggregated memory is a much smaller and simpler trusted computing base. Moreover, the fast/slow path approach of our implementation allows it to operate fast in the common case by entirely bypassing signatures and the trusted computing base. This section quantifies the performance of our implementation of CTBcast against the de facto way of preventing equivocation in modern systems [20, 122, 172], namely a trusted counter implemented on Intel's SGX.

Non-equivocation mechanisms based on trusted counters work by securely binding a monotonically increasing sequence number to each message broadcast by a process. All enclaves store a local *counter* and share a common *secret*. Before broadcasting a message, a process feeds it to its local enclave and gets back a proof of non-equivocation of the form

$HMAC_{secret}(msg||counter||process\ id)$. The recipients of a message use their own local enclave to verify the authenticity of the HMAC, and thus prevent equivocation. Note that, as authentication takes place exclusively in the enclaves, this non-equivocation mechanism avoids the use of expensive public-key cryptography.

Figure 5.10 shows the median latency of the two non-equivocation mechanisms between a sender and two receivers. The lowest latency is achieved by the fast path of our implementation of CTBcast and ranges from 2.2 µs to 11 µs depending on the message size. In this mode of operation, CTBcast delivers on unanimous and timely participation of all processes to avoid signatures and prevent equivocation using merely two rounds of Tail Broadcast. CTBcast's slow path (triggered under failures) relies on public-key cryptography, which dominates the latency and raises it to approximately 86 µs. The latency of preventing equivocation using SGX includes accessing the enclave twice, once in the sender and once in the receiver, as well as broadcasting the message.

Due to the lack of SGX in our RDMA experimental setup, we evaluate the cost of accessing the SGX on different hardware to get a good approximation. The SGX hardware is provided by a machine with an Intel i7-7700K CPU running at 4.2 GHz, which is 0.6 GHz higher than the CPU frequency of our RDMA setup. Accessing the enclave (once) ranges from 7 µs to 12.5 µs, which makes preventing equivocation using the SGX take at least 16 µs. The resulting latency is shown in the middle line of Figure 5.10.

Overall, for both non-equivocation approaches, larger message sizes lead to a linearly higher latency, due to the hashing and communication latency. CTBcast's fast path is up to 6.5× faster than the SGX solution, by taking advantage of the ultra fast communication.

5.7.5 Impact of CTBcast's Tail on Tail Latency

Figure 5.11 shows how the size of the tail in CTBcast (parameter t) affects the client's tail latency. We focus on uBFT's fast path and execute Flip with small 64 B requests and larger 2 KiB ones. For both request sizes, we explore four *tail* parameters.

For smaller values of t, we see a latency spike indicative of thrashing as we move to higher percentiles. This spike occurs because CTBcast uses a double buffering mechanism with cryptographic summaries (§5.5.2) to switch between them; if both buffers fill before a summary occurs (due to a small t), CTBcast stalls. The smaller the t, the sooner the buffers fill, the more often CTBcast stalls, and hence the lower the percentile of the spike. For small requests, a tail t=128 avoids thrashing up to the 99th percentile. For larger requests, t=64 suffices, as filling the buffers takes more time, giving more time for the summary to occur.

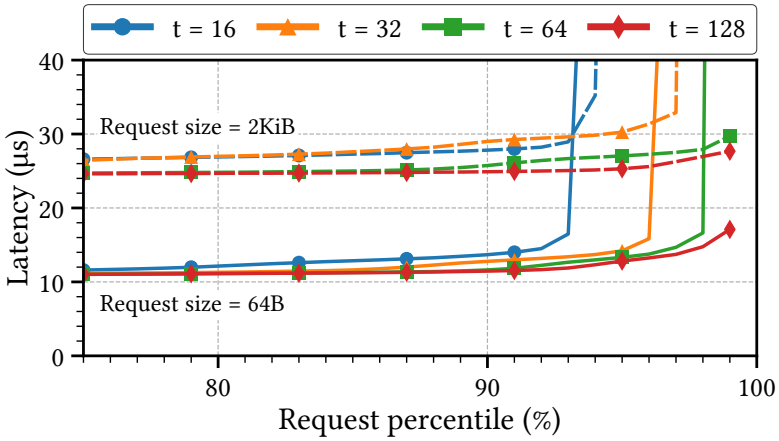

Figure 5.11: uBFT's tail latency for different CTBcast tails t for 2 KiB requests (top) and 64 B requests (bottom).

5.7.6 Memory Consumption

Given the fundamental goal of uBFT to operate using finite memory, we monitor the consumption of disaggregated memory and the local memory consumption at the leader replica, while re-running the experiment of Section 5.7.5.

Table 5.2 summarizes the results for two different request sizes (64 B and 2 KiB) and four different $tail$ parameters of CTBcast (16, 32, 64 and 128). For the small 64 B requests, the local memory consumption starts at 0.46 GiB. This is the entire memory that uBFT preallocates at startup and uses during its lifetime. When CTBcast's tail t increases, uBFT's memory consumption increases linearly by ≈1 MiB for each additional message in the tail. For the large 2 KiB requests, the memory consumption starts at 4.3 GiB (t=16) and increases at a rate of ≈11 MiB per message.

uBFT consumes little disaggregated memory. The last row of Table 5.2 shows the memory used at a single memory node. This amount is independent of the size of requests and depends only on CTBcast's tail t: messages sent over CTBcast are transmitted using our fast message-passing primitive; upon receiving a message, a receiver writes to disaggregated memory only the message's id and its fingerprint (a 32 B cryptographic hash); the register implementation (§5.6.1) stores two copies, each with an 8 B checksum. To save space, registers

Table 5.2: uBFT replica (top) and disaggregated (bottom) memory usage for different CTBcast tails t and request sizes.

Request size	**t = 16**	**t = 32**	**t = 64**	**t = 128**
64 B	0.46 GiB	0.47 GiB	0.49 GiB	0.53 GiB
2 KiB	4.3 GiB	4.5 GiB	4.8 GiB	5.5 GiB
Disag. Mem.	20 KiB	40 KiB	81 KiB	162 KiB

use the identifiers of messages as their timestamps.

5.8 Related Work

uBFT uses RDMA to instantiate disaggregated memory. Prior work identifies some benefits and downsides of using RDMA in an adversarial environment. Aguilera *et al.* [4] propose new RDMA-based techniques to enhance the resilience and performance of BFT algorithms. That work is abstract and far from practical solutions: it requires infinite memory and solves only single-shot consensus, stopping short of a solution for an SMR system. Rüsch *et al.* [154] design a Byzantine fault tolerant system that uses RDMA to improve performance, but requires $3f+1$ processes.

Prior work identifies vulnerabilities in the current generation of RDMA hardware and proposes ways to mitigate them [153, 165]. That work is orthogonal to uBFT and could be applied to it. We expect that these problems will eventually be fixed with future NICs.

With a black-box mechanism to prevent equivocation, only $2f+1$ replicas are required for BFT [21, 22, 45]. Several BFT systems achieve that using trusted hardware as the black box: attested append-only memory (A2M) [44] uses a trusted log, TrInc [120] and MinBFT [172] use a trusted counter, Hybster [20] uses Intel's SGX, CheapBFT [93] uses FPGAs, and H-MFT [182] uses trusted hypervisors to implement write-once tables. By separating execution from agreement [178], one can reduce the number of execution replicas to $2f+1$, but $3f+1$ replicas are still required for agreement. Avocado [17] implements a high-performance replicated confidential KV-store using CPU enclaves as the trusted computing base.

Blockchain systems also tolerate Byzantine failures, but their latency is in the seconds or minutes due to their heavy use of cryptography [71, 100], proof of work [67, 137], and/or batching [33, 179]. The recent SplitBFT [129] uses SGX and $3f+1$ replicas to strengthen the safety and confidentiality of blockchains in public clouds.

While most of the prior work is not focused on microsecond-scale latency (and hence came up with different solutions from ours), some recent SMR systems aim for lower latency. Mu [5] is highly optimized and provides microsecond-scale performance, but tolerates only crash failures. SBFT [73] tolerates Byzantine failures and uses a fast path to improve latency, but does not achieve microsecond-scale performance due to its use of cryptography. BFT SMR systems with $3f+1$ replicas can avoid cryptographic signatures, for example, in PBFT's optimized implementations [37]. However, this is impossible in a system with $2f+1$ replicas [45]; the key to uBFT's performance is thus avoiding signatures on the fast path.

Carbink [181] and Hydra [115] build reliable disaggregated memory to improve memory utilization in a cluster, albeit without support for concurrent shared access. MIND [114], GAM [36] and Clover [170], on the other hand, provide reliable shared memory, but they do not tolerate Byzantine writers.

5.9 Discussion

Does microsecond 2f+1 BFT require disaggregated memory? To achieve microsecond-scale BFT SMR, one must avoid the use of expensive signatures and trusted components in the critical path. uBFT does so via a fast/slow path design that uses disaggregated memory in the slow path. This leads to a small trusted computing base, but there may be other ways to achieve non-equivocation in CTBcast's slow path without affecting fast-path performance.

Can uBFT work with a Byzantine network? uBFT assumes network connections are authenticated and tamper-proof, which is realistic in data centers, where widely deployed protocols such as IPsec and SSL provide such guarantees at line rate. What if such protocols are not available? We can implement simple authenticated channels within uBFT without signatures in the critical path, by augmenting messages with an HMAC. With BLAKE3, creating or verifying 256-bits HMACs takes ≈100 ns. As a result, we expect less than 2 µs of additional overhead on the fast path of uBFT.

What about uBFT's throughput? Any system can provide a throughput that is inverse of its latency. For uBFT, that amounts to ≈91 kops for small 32 B requests. uBFT doubles this figure, by exploiting the slack between the processing of events in a consensus slot to interleave two requests with minimal latency penalty. Throughput can be further optimized using well-known techniques, such as batching [53] and running parallel consensus instances on multiple cores [20], but we have not done so.

Epilogue Part IV

6 Concluding Remarks

We conclude this thesis by summarizing its main findings and their implications, as well as by outlining promising directions for future work.

6.1 Summary and Implications

At a high level, we believe that this thesis takes a step forward towards a better understanding of reliable microsecond-scale distributed computing. The value of this endeavor is not just the performance figures. It is showing that even though there is not enough leeway at this time scale, it is still possible to achieve low latency without sacrificing reliability. To achieve this, we had to move between theory and practice: come up with inventive ways of using modern networking hardware, go back to theoretical analysis to assess the feasibility of our ideas, and then adapt the theory to the practical setting by following a pragmatic, rather than a puristic, approach. We hope that programmers and practitioners can use the insights and ideas in this thesis to build systems that meet the needs of modern data center applications.

The thesis was structured around two antipodal types of failures, crash-stop and Byzantine, which we approach systematically and methodically in Parts II and III, respectively. We believe that fault-tolerance is a crucial property of any application that powers important infrastructure, which is what many data center applications have become.

In Part II, we focused on crash-stop failures and put our efforts on devising a generic mechanism for handling them. This mechanism is the membership service, and though such services existed before (e.g., ZooKeeper), none of them was suitable for the microsecond scale. For this reason, we built uKharon to allow microsecond-scale applications escape ad-hoc solutions and attain dynamicity with minimal latency penalty. In fact, we exposed dynamicity through a simple and reusable interface and showed how this interface can be used to easily build fault-tolerant applications.

In Part III, we focused on Byzantine failures. Such shift in paradigm is motivated by empirical evidence that data center operators observe spurious, unpredictable events that can lead to

malfunction of their systems and potentially high financial loss. Once again, with genericity in mind, we opted for state machine replication (SMR) in order to provide programmers with an easy way towards Byzantine fault tolerance. Handling Byzantine failures, however, is traditionally expensive due to the increased number of replicas ($3f+1$ instead of $2f+1$) or the presence of digital signatures. Thus, we set out to study the inherent cost of Byzantine failures when using $2f+1$ replicas, and assess whether shared memory provided by modern networking hardware could help avoid digital signatures. We theoretically proved that signatures are necessary, yet we devised algorithms that avoided them most of the time, and assessed their optimality. We then continued by building an entire system based on these ideas, only to find another major obstacle: infinite memory consumption. Thankfully, we worked around it by adding a few extra signatures. The end result is a system, called uBFT, that is the first to achieve microsecond-scale Byzantine fault tolerant SMR. We believe that with it, data center operators can be assured of resilience to various unforeseen events, thanks to the increased, yet low-cost, reliability offered by uBFT.

6.2 Future Directions

Real-Time Execution. In Chapter 3, we show that fast failure detection is essential for membership services operating at the microsecond scale. The middle level of our multi-level failure detector depends on its timely behavior and should be impervious to execution fluctuations, such as kernel jitter. However, the Linux kernel does offer strong guarantees for real-time execution. Patches that enhance the real-time behavior of the Linux kernel (e.g., tickless scheduling of processes) already exist, but they reduce rather than eliminate unpredictability of execution. We believe that as microsecond-scale computing becomes ubiquitous, the need for strict real-timeness in the software stack supporting microsecond-scale applications will become more relevant. This opens the question of whether it is possible to co-design new hardware and software with strong real-time guarantees for microsecond-scale applications.

Broadcast Abstraction for Consensus. In Chapter 4, we designed a $2f+1$ Byzantine consensus protocol that uses Consistent Broadcast instead of Reliable Broadcast. As a result, the final protocol resulted in fewer signatures than existing consensus protocols based on Reliable Broadcast. The intuitive explanation for this is that sometimes abstractions pack more than what is exactly needed to solve the problem at hand. From a theoretical standpoint, it is interesting to study whether we can define a weaker primitive than Consistent Broadcast that is ideally fitted to this problem.

State Transfer. In Chapters 3 and 5, we looked at replication. A critical part of replication is the ability to efficiently bring replicas up to speed, a procedure known as *state transfer*. Typically, applications perform state transfer by incorporating specific support for it; effectively the application can take a snapshot of itself. However, this approach increases application

complexity, especially in the face of concurrency, and is application specific. We believe that application-agnostic state transfer is an open problem. In principle, migration techniques devised for virtual machines (VMs) could help in snapshotting an application, especially when considering crash-stop failures. However, with Byzantine failures the situation is more difficult, as VM-style snapshots contain a lot of internal state (e.g., threads, garbage collector state) which one has to somehow validate before adopting the snapshot.

Faster Signatures. In Chapters 4 and 5, we identified signatures as the main reason for why Byzantine fault tolerance is typically avoided in the microsecond scale. However, this observation is based on existing signature schemes, leaving the open question whether we could devise schemes that could benefit from hardware acceleration (e.g., GPUs, or FPGAs), to make signatures faster than ever. We believe that such accomplishment is not just wishful thinking, as there exist signature schemes (e.g., SPHINCS [24]) that utilise exclusively cryptographic hashes to become quantum resistant. Cryptographic hashes are easy to compute and can be parallelized. The question is whether we could adapt such schemes to leverage the hardware present in the data centers.

7 Appendices

A Appendix for uKharon

A.1 Correctness of One-Sided Paxos

A.1.1 Assumptions

In the next subsections, we consider the M&M model [3]. It allows processes to both pass messages and share memory. We assume that communication channels are lossless and have FIFO semantics, which is ensured by InfiniBand's Reliable Connections. The system has n processes $\Pi = \{p_1, \ldots, p_n\}$ that can attain the roles of *proposer* or *acceptor*. There are p proposers and n acceptors. Up to $p-1$ proposers and $\lfloor \frac{n-1}{2} \rfloor$ acceptors may fail by crashing. As long as a process is alive, its memory is remotely accessible. When a process crashes, subsequent operations to its memory hang forever. We assume partial synchrony for consensus's liveness [70].

A.1.2 One-Sided RPC

In this section, we prove that the one-sided RPCs of Algorithm 3.1 are equivalent to two-sided RPCs when not obstructed. Moreover, we prove that when equivalence is violated (due to obstruction), one-sided RPCs have no side effects. We assume that both `compare` and `f` are deterministic.

Lemma A.1.1. *If `cas-rpc` does not abort, `rpc` and `cas-rpc` are equivalent.*

Proof. An execution of `rpc` solely depends on the value of `state` and the input value `x`. We denote such execution of `rpc` as $\langle state, x \rangle_{rpc}$. If an execution of `cas-rpc` does not abort, it solely depends on the value of `expected` fetched at line 2 and the input value `x`. We denote such execution of `cas-rpc` as $\langle expected, x \rangle_{cas-rpc}$.

We show that any execution $\langle s, x \rangle_{rpc}$ is equivalent to the execution $\langle s, x \rangle_{cas-rpc}$ in the sense that both `rpc` and `cas-rpc` will have the same `state` value and return the same projection at the end of their execution.

If an execution $\langle s_1, x\rangle_{rpc}$ makes the comparison at line 3 fail, then `state` is not modified and `proj(`s_1`)` is returned. In the execution $\langle s_1, x\rangle_{cas-rpc}$, the comparison at line 3 will also fail and `proj(`s_1`)` is also returned without modifying the remote state. In this case, both executions are equivalent.

If an execution $\langle s_2, x\rangle_{rpc}$ makes the comparison at line 3 succeed, then `state` is modified to `f(`s_2`, x)` and `proj(f(`s_2`, x))` is returned. In the execution $\langle s_2, x\rangle_{cas-rpc}$, the comparison at line 3 will also succeed. As the execution is assumed not to abort, the CAS will succeed. Thus the remote state will atomically be updated from s_2 to `f(`s_2`, x)` and `proj(f(`s_2`, x))` is also returned. In this case, both executions are also equivalent. □

Lemma A.1.2. *If `cas-rpc` aborts, it has no side effects.*

Proof. If `cas-rpc` aborts, the comparison at line 7 has failed. This implies that the CAS failed and thus that `state` is unaffected by the execution. □

From lemmas A.1.1 and A.1.2, `cas-rpc` exhibits all-or-nothing atomicity. We now prove that such a transformation is obstruction-free.

Lemma A.1.3. *If `cas-rpc` runs alone, it does not abort.*

Proof. Let's assume by contradiction that `cas-rpc` runs alone and aborts. For `cas-rpc` to abort, the comparison at line 7 must have failed. This implies that the CAS at line 6 failed due to `state` not matching `expected`. `state` must thus have been updated between lines 2 and 6. This implies a concurrent execution, hence a contradiction. □

A.1.3 Consensus and Abortable Consensus

In the consensus problem, processes *propose* individual values and eventually irrevocably *decide* on one of them. Formally, consensus has the following properties:

Termination Every correct process eventually decides once.

Uniform agreement If v and v' are decided on, then $v = v'$.

Validity If v is decided on, v is the input of some process.

We implement consensus by composing two abstractions:

- *Abortable consensus* [34], an abstraction weaker than consensus that is solvable in the asynchronous model,
- *Eventually perfect leader election* [41], the weakest failure detector required to solve consensus.

Abortable consensus is identical to consensus except for:

Termination Every correct process eventually decides once or aborts.

Decision If a single process proposes infinitely many times, it eventually decides.

Algorithm A.1: Paxos's Abortable Core

```
1  # At proposers:
2  decided = False
3  proposal = id
4  proposed_value = ⊥

6  def propose(value):
7    proposed_value = value
8    prepare()
9    accept()

11 def prepare():
12   proposal = proposal + |Π|
13   broadcast ⟨Prepare | proposal⟩
14   wait for a majority of ⟨Prepared | ack, ap, av⟩
15   adopt av with highest ap as proposed_value
16   if any not ack: abort

18 def accept():
19   broadcast ⟨Accept | proposal, proposed_value⟩
20   wait for a majority of ⟨Accepted | mp⟩
21   if any mp > proposal: abort
22   trigger once ⟨Decide | proposed_value⟩

24 # At acceptors:
25 min_proposal = 0
26 accepted_proposal = 0
27 accepted_value = ⊥

29 upon ⟨Prepare | proposal⟩:
30   if proposal > min_proposal: min_proposal = n
31   reply ⟨Prepared | min_proposal == n, accepted_proposal, accepted_value⟩

33 upon ⟨Accept | proposal, value⟩:
34   if proposal ≥ min_proposal:
35     accepted_proposal = min_proposal = n
36     accepted_value = value
37   reply ⟨Accepted | min_proposal⟩
```

A.1.4 One-Sided Abortable Consensus

Algorithm A.1 appears in [34] and implements abortable consensus Algorithm A.2 transforms algorithm A.1 by replacing its RPCs with CAS-based RPCs. This transformation causes it to abort strictly more than the original algorithm. To see why, consider the following execution: Let proposers P_1 and P_2 concurrently initiate the Prepare phase with respective proposals 1

Algorithm A.2: One-Sided Abortable Consensus

```
1  # At acceptors:
2  state = { min_proposal: 0, accepted_proposal: 0, accepted_value: ⊥}

4  # At proposers:
5  proposal = id
6  proposed_value = ⊥

8  def propose(value):
9    proposed_value = value
10   prepare()
11   accept()

13 def prepare():
14   proposal = proposal + |Π|
15   async cas_prepare(p) for p in Acceptors
16   wait for a majority to return ⟨ack, ap, av⟩
17   if any not ack: abort
18   adopt av with highest ap as proposed_value

20 def accept():
21   async cas_accept(p) for p in Acceptors
22   wait for a majority to return mp
23   if any mp > proposal: abort
24   trigger once ⟨Decide | proposed_value⟩

26 def cas_prepare(p):
27   expected = fetch_state(p)
28   if not proposal > expected.min_proposal:
29     return ⟨False, expected.accepted_proposal, expected.accepted_value⟩
30   move_to = expected
31   move_to.min_proposal = proposal
32   read = state_p.cas(expected, move_to)
33   if read == expected:
34     return ⟨True, expected.accepted_proposal, expected.accepted_value⟩
35   abort

37 def cas_accept(p):
38   expected = fetch_state(p)
39   if not proposal ≥ expected.min_proposal:
40     return expected.min_proposal
41   move_to = expected
42   move_to.min_proposal = proposal
43   move_to.accepted_proposal = proposal
44   move_to.accepted_value = proposed_value
45   read = state_p.cas(expected, move_to)
46   if read == expected:
47     return expected.min_proposal
48   abort
```

and 2. Both fetch the remote state and get $\langle 0,0,\perp\rangle$. Then, P_1 succeeds in writing its proposal to acceptor A_1. Later on, the CAS of P_2 fails at A_1 as the value is now $\langle 1,0,\perp\rangle$ instead of the expected $\langle 0,0,\perp\rangle$. Thus, P_2 aborts even if it had a larger proposal number than P_1. The more relaxed comparison in the original algorithm would not have caused P_2 to abort.

Lemma A.1.4. *Algorithm A.2 preserves Decision.*

Proof. If a single process proposes infinitely many times, it will eventually run the one-sided RPCs obstruction-free. By Lemma A.1.3, this guarantees that the one-sided RPCs will eventually terminate without aborting. In such case, Lemma A.1.1 guarantees the execution to be equivalent to one of the original algorithm. Thus, the transformation preserves the decision property of Algorithm A.1. □

Lemma A.1.5. *Algorithm A.2 preserves Termination.*

Proof. Assuming a majority of correct acceptors, CASes will eventually complete at a majority. Due to the absence of loops or blocking operations inside `prepare`, `accept`, `cas_prepare` and `cas_accept` in algorithm A.2 (apart from waiting for the completion of CASes at a majority), a proposer that invokes `propose` will either abort or decide. □

Algorithms A.1 and A.2 differ only in some executions where the transformed algorithm aborts whereas the original does not. Nevertheless, aborting does not violate safety, as we show next.

Lemma A.1.6. *Algorithm A.2 preserves the safety properties.*

Proof. Assume, by contradiction, that adding superfluous abortions in Algorithm A.1 violates safety. Consider an execution E_1, where processes $\{P_1, ..., P_n\}$ deviate from the algorithm and abort at times $\{t_1, ..., t_n\}$ after which the global state is $\{S_1, ..., S_n\}$ and safety is violated. Also, consider another execution E_2, where processes $\{P_1, ..., P_n\}$ crash at times $\{t_1, ..., t_n\}$ after which the global state is $\{S_1, ..., S_n\}$. In execution E_1, safety is violated. On the other hand, execution E_2 preserves safety, since Algorithm A.1 tolerates arbitrarily many proposer crashing. The two executions, however, are indistinguishable, hence a contradiction. Thus, Algorithm A.2 preserves safety regardless of how often it aborts. □

Theorem A.1.7. *Algorithm A.2 implements abortable consensus.*

Proof. The result follows directly by composing lemmas A.1.4, A.1.5 and A.1.6. □

A.1.5 Streamlined One-Sided Algorithm

In this section, we make Algorithm A.2 efficient in order to increase its practicality.

First, it is not required to fetch the remote state at the start of each RPC. As it is safe to have stale `expected` states, it is safe to use states deduced from previous CASes. Predicted states can thus be initialized to $\langle 0, 0, \perp \rangle$ and updated each time a CAS completes (either succeeding or not). Moreover, wrongly predicting states can only result in superfluous aborts which have been proven to be safe by Lemma A.1.6. Thus, it is safe to optimistically assume that onflight CASes

will succeed. Second, in the Prepare phase, the `proposal` variable can be increased upfront to value higher than any predicted remote `min_proposal` to reduce predictable abortions.

Algorithm A.3: Streamlined One-Sided Abortable Consensus

```
1  # At acceptors:
2  state = { min_proposal: 0, accepted_proposal: 0, accepted_value: ⊥}
3
4  # At proposers:
5  predicted[] = { 0, 0, ⊥}
6  proposal = id
7  proposed_value = ⊥
8
9  def propose(value):
10   proposed_value = value
11   prepare()
12   accept()
13
14 def prepare():
15   while any predicted[.].min_proposal ≥ proposal:
16     proposal = proposal + |Π|
17   for p in Acceptors:
18     move_to[p] = {min_proposal: proposal, ..predicted[p]}
19     reads[p] = async state_p.cas(predicted[p], move_to[p])
20   wait until majority of states are read
21   for p in Acceptors:
22     if reads[p] ∈ {predicted[p], ⊥}:
23       predicted[p] = move_to[p]
24     else:
25       predicted[p] = reads[p]
26   if any CAS failed: abort
27   adopt proposed_value from predicted accepted_values with highest accepted_proposal
       ↪ if any
28
29 def accept():
30   reads = ⊥^|Acceptors|
31   move_to = (proposal, proposal, proposed_value)
32   for p in Acceptors:
33     reads[p] = async state_p.cas(predicted[p], move_to)
34   wait until majority of states are read
35   if any CAS failed:
36     for p in Acceptors:
37       if reads[p] ∈ {predicted[p], ⊥}:
38         predicted[p] = move_to
39       else:
40         predicted[p] = reads[p]
41     abort
42   trigger once ⟨Decide | proposed_value⟩
```

With the aforementioned optimisations, Algorithm A.2 is transformed into Algorithm A.3. Notably, the liveness of the resulting algorithm is preserved: Let's assume that a *single* proposer runs infinitely many times. Eventually, it will run obstruction-free. In the worst case, each time it will abort at line 26 or 41 because of a single wrong guess and update its prediction. The optimistic update of expected states at lines 23 and 38 and the FIFO semantics of commu-

nication links provide that, once a remote state is correctly guessed, any later CAS will succeed. Thus, after at most n runs, all CASes will succeed and the proposer will decide.

A.2 Correctness of the Active Method

In this section, we provide a formal definition and a proof of correctness of the `Active` method described in Section 3.6.

A.2.1 Formal Definition

`Active(`*Membership*`)` $\rightarrow$ *bool* has the following properties:

Monotonicity If `Active(M')` returns `true` at any process, future calls `Active(M)` with $M < M'$ will return `false`.

Convergence If M is the last membership to be decided (if any), invoking `Active(M)` will eventually return `true` at all correct processes.

Definition 4. *If `Active(M)` returns `true`, then M is considered active at the linearization point of the call.*

Definition 5. *If M is active at times t and t', then it is considered active in the interval $[t, t']$.*

From these simple properties and definitions, it follows that no two active memberships can overlap.

Theorem A.2.1. *Only one membership can be active at a time.*

Proof. Assume by contradiction that M and M' ($M < M'$) are simultaneously active. By definition, `Active(M)` must have returned `true` after `Active(M')` returned `true`. This breaks Monotonicity, hence a contradiction. □

A.2.2 Non-Leased Active Membership

We prove the correctness of uKharon's implementation of `Active`. We assume no gaps in the sequence of decided memberships. This is enforced by coordinators by not proposing the $(k+1)$-th membership until the k-th is decided.

Lemma A.2.2. *Algorithm A.4 ensures Monotonicity.*

Proof. `Active` can only be called on decided memberships. Let M and M' be two decided memberships with $M < M'$. If `Active(M')` returned `true`, by the no-gap assumption, all

Algorithm A.4: Active built on top of the consensus engine

```
1  def Active(M) → bool:
2    reads = ⊥^|Acceptors|
3    for p in Acceptors:
4      reads[p] = async paxos[M.id + 1].slot_p.read()
5    wait until majority of slots are read
6    if all slots are not accepted:
7      return true
8    propose_membership(M.id + 1, first accepted value)
9    return false
```

memberships between M and M' have been decided. Because M's successor has been decided, a majority of acceptors' slots `M.id + 1` have been written. Thus, `Active(M)` will read at least one non-empty slot and return `false`. □

Lemma A.2.3. *Algorithm A.4 ensures Convergence.*

Proof. Assume by contradiction that M is the last decided membership and `Active(M)` never returns `true` at some correct process. Thus, this process executes line 8, which means that it proposes a new membership. Given that the process is correct, some membership with id `M.id + 1` will eventually be decided. Therefore, M is not the last membership, hence a contradiction. □

Theorem A.2.4. *Algorithm A.4 implements `Active`.*

Proof. Follows directly from Lemmas A.2.2 and A.2.3. □

A.2.3 Leased Active Membership

Algorithm 3.2 reduces communication by leasing the output of Algorithm A.4. We prove that it preserves `Active`'s properties.

Lemma A.2.5. *Algorithm 3.2 preserves Monotonicity.*

Proof. Let e be an execution of `Active(M)` that returned `true`. e either returned at line 9 or at line 12 with `t > t`$_{start}$. We denote the former case $leased(M)$ and the latter $checked(M)$. Assume by contradiction that `Active(M')` returned `true` in an execution e_1 and then `Active(M)` returned `true` in an execution e_2 with $M < M'$. Either:

- $leased(M)$: In e_2, `majority_active(M)` returned `true` at most δ before `Active(M)` returned `true`. In e_1, lines 5–7 ensure that M' was decided at least δ before `Active(M')` returned `true`. Thus, `majority_active(M)` returned `true` after M' was decided. However, because M' has been decided, a majority of acceptors' slots `M'.id = M.id + 1`

must have been written. Thus, `majority_active(M)` should have read at least one non-empty slot and returned `false`. Hence, a contradiction.

- *checked*(*M*): `majority_active(M)` returned `true` after `majority_active(M')` returned `true`. This breaks `majority_active`'s Monotonicity, hence a contradiction.

□

Lemma A.2.6. *Algorithm 3.2 preserves Convergence.*

Proof. Assume that M is the last membership to be decided. Thus, `majority_active(M)` will eventually always return `true`. At most δ after `Active(M)` returns for the first time, t_{start} will be in the past and `leased_membership` set to M. Thus, eventually, the `else` branch at line 8 will always be visited and either return $true$ via line 9 or 12. □

Theorem A.2.7. *Algorithm 3.2 implements `Active`.*

Proof. Follows directly from Lemmas A.2.5 and A.2.6. □

A.3 Clocks

uKharon relies on hardware timestamps to check if a membership is `Active`. When using modern Intel processors, Linux has three available clocksources: `tsc`, `hpet` and `acpi_pm`. The `tsc` clocksource is the most efficient and requires 20-25ns to take a timestamp [63].

Architectural considerations. The `tsc` clocksource uses Intel's TSC hardware to measure time accurately. TSC stores the number of cycles executed by the CPU after the latest reset. Traditionally, TSC is considered an unreliable way to take timestamps. The reason is that Intel processors have variable clock speed, thus the number of cycles does not correspond to wallclock time. However, modern Intel processors have three features [48]: *Constant TSC*, *Nonstop TSC* and *Invariant TSC* which solve this problem. The combination of these features results in a TSC that is incremented at a constant rate regardless of the power state of the processor. As a result, it is safe to use this counter for efficient timestamping.

TSC synchrony. In Intel processors, every core has its own TSC. All processors in the same socket start the TSC hardware using the same RESET signal, thus the absolute values of the TSC across cores of the same socket match. This means that one can compare safely the values of TSC across different cores, assuming that all TSCs run at the same frequency. Because this assumption does not always hold, Linux determines the base frequency of every core during boot and uses this frequency to convert clock cycles to wallclock time. To accomplish it, Linux uses the more accurate (and more expensive) `hpet`.

uKharon takes further care to deal with TSC synchrony. More precisely, it checks for the synchronization of TSC between cores using a ping-pong test. In this test, core A takes a timestamp t_1 and signals core B to do the same. Core A signals core B by writing to a lock-free Single-Producer Single-Consumer (SPSC) queue that is polled by B. When B receives the signal it also takes a timestamp t_2 and sends it back to A (using another SPSC queue). Upon reception of the timestamp from B, core A takes the last timestamp t_3. In our test we confirm that always $t_1 < t_2 < t_3$. Additionally, in our hardware, the minimum difference between t_1 and t_2 is $\epsilon = \min(t_2 - t_1)$ is 64ns. uKharon takes ϵ into consideration by incorporating into the leases as follows: Suppose a lease is valid for a duration of δ starting at time t. uKharon considers that the lease starts at time $t + \epsilon$ and has a duration of $t + d - 2\epsilon$.

Inter-machine clock drift. In order to ensure that active memberships do not overlap, uKharon assumes that clock drift is bounded, i.e., that time passes approximately at the same speed on different machines. This assumption is necessary to enable client-side leases. It guarantees that after a lease duration period, leases across all clients will have to be renewed. Our system is built to tolerate clock drift, as long as this drift is bounded. We experimentally determine an upper bound for the clock drift with a simple test. In this test, machine A takes a timestamp t_1 and pings machine B to wait for 1 minute before replying back to it. Upon reception of B's response, A takes another timestamp t_2. It then computes $t_2 - t_1$ and compares it to the expected 1 minute measured by B (after removing the communication delay). We repeat this test several times and determine that the clock drift between machines differs by at most 0.001%. uKharon incorporates inter-machine clock drift by waiting $1.01 \times \delta$ upon membership discovery, ensuring that when leases become active on a new membership, everyone's leases on the previous membership will have expired.

B Appendix for Frugal Byzantine Computing

B.1 Correctness of Consistent Broadcast

We start with a simple observation:

Observation 1. *If p is a correct process, then no sub-slot that belongs to p is written to more than once.*

Proof. Since p is correct, p never writes to any sub-slot more than once. Furthermore, since all sub-slots are single-writer registers, no other process can write to these sub-slots. □

Before proving that our implementation, Algorithm 4.1, satisfies the properties of Consistent Broadcast, we show two intermediary results with respect to the liveness and safety of the scan operation.

Lemma B.1.1 (Termination of scan.)**.** *Each scan operation returns.*

Proof. We observe the following:

1. If some slot S goes from empty to non-empty between consecutive iterations of the while loop, then some process (the writer of slot S) wrote a value to S. This causes the while loop to continue (line 39).
2. Termination condition: the while loop exits (and thus the *scan*() operation returns) once either (a) *others* has no empty slots, or (b) no slot goes from empty to non-empty between two consecutive iterations of the while loop (line 39 is never executed).
3. Once a slot S is read and has non-empty value, *others* gets updated (lines 31 or 37) and the slot S is never read again.
4. the size of *others* is equal to the number of processes, n.

By contradiction, assume the scan function never terminates. This implies the scan function did not return after executing $n+1$ iterations of the while loop. This means, each iteration at least one slot went from being empty to containing a value. This value is added to the *others* array (line 37) and the slot is not read again in future iterations. Since the array *others* is bounded by n, after n iterations *others* contains only non-empty values. At the $n+1^{th}$ execution of the loop, no slot is empty, *done* stays true, and hence the operation returns. Contradiction.

Given every scan operation returns after executing at most $n+1$ times the while loop and each loop invokes at most n reads of the base registers, the complexity of the *scan*() operation is within $O(n^2)$. □

Lemma B.1.2 (Non-inversion of scan)**.** *Let p_1 and p_2 be correct processes who invoke scan. Let V_1 and V_2 be the return values of those scans, respectively. If V_1 contains some value m_1 in at least $n-f$ slots and V_2 contains some value m_2 in at least $n-f$ slots, then V_1 contains m_2 in at least one slot, or V_2 contains m_1 in at least one slot.*

Proof. Assume by contradiction that V_1 does not contain m_2 and V_2 does not contain m_1.

Since V_1 contains m_1 in at least $n-f$ slots, it must be that m_1 was written by at least one correct process; call this process r_1. Similarly, m_2 must have been written by at least one correct process; call this process r_2. Then the following must be true:

1. p_1 must have read the slot of r_2 at least twice and found it empty. Let $t_{p_1 \leftarrow r_2}$ be the linearization point of p_1's last read of r_2's slot before p_1 returns from the scan.
2. p_2 must have read the slot of r_1 at least twice and found it empty. Let $t_{p_2 \leftarrow r_1}$ be the linearization point of p_2's last read of r_1's slot before p_2 returns from the scan.
3. p_1 must have read the slot of r_1 and found it to contain m_1. Let $t_{p_1 \leftarrow r_1}$ be the linearization point of p_1's last read of r_1's slot.
4. p_2 must have read the slot of r_2 and found it to contain m_2. Let $t_{p_2 \leftarrow r_2}$ be the linearization point of p_2's last read of r_2's slot.

We now reason about the ordering of $t_{p_1 \leftarrow r_1}$, $t_{p_1 \leftarrow r_2}$, $t_{p_2 \leftarrow r_1}$, and $t_{p_2 \leftarrow r_2}$:

1. $t_{p_1 \leftarrow r_1} < t_{p_1 \leftarrow r_2}$. Process p_1's last read of r_2's slot must have occurred during the last iteration of the while loop before returning from the scan. Furthermore, p_1's last read of r_1's slot cannot have occurred on the same last iteration of the loop, otherwise the non-empty read would have triggered another iteration; thus, p_1's last read of r_1's slot must have occurred either in a previous iteration at line 37, or initially at line 31.
2. Similarly, $t_{p_2 \leftarrow r_2} < t_{p_2 \leftarrow r_1}$.

3. $t_{p_1 \leftarrow r_2} < t_{p_2 \leftarrow r_2}$. Process p_1's last read of r_2 returns an empty value, while p_2's last read of r_2 returns m_2. Since r_2 is correct, its slot cannot go from non-empty to empty, thus the empty read must precede the non-empty read.

4. Similarly, $t_{p_2 \leftarrow r_1} < t_{p_1 \leftarrow r_1}$.

By transitivity, from (1)-(3), it must be that $t_{p_1 \leftarrow r_1} < t_{p_2 \leftarrow r_1}$. This contradicts (4). No valid linearization order exists for the four reads. □

We are now ready to prove that Algorithm 4.1 satisfies the properties of Consistent Broadcast.

Lemma B.1.3 (Validity)**.** *If a correct process s broadcasts m, then every correct process eventually delivers m.*

Proof. Let s be a correct sender that broadcasts m and consider a correct receiver p that tries to deliver s's message.

Since s is correct, it writes m to its message sub-slot. Therefore, all replicators read m and no other message from s, by Observation 1.

If all replicators are correct and they copy m in a timely manner, then p is able to deliver m via the fast path (at line 25).

Otherwise s, being correct, will eventually write its (valid) signature of m to its signature sub-slot. Since we consider at most f Byzantine processes that replicate m, the $n-f$ correct replicators are guaranteed to copy the signature of m to their slot. Moreover, since s is correct, and we assume Byzantine processes cannot forge the digital signatures of correct processes, no replicator can produce a different message $m' \neq m$ with s's signature. This enables receiver p to deliver m via the slow path (at line 28). □

Lemma B.1.4 (No duplication)**.** *Every correct process delivers at most one message.*

Proof. Correct processes only deliver at lines 25 or 28. Immediately after a correct p process delivers a message, p exits the *while* loop and thus will not deliver again. □

Lemma B.1.5 (Consistency)**.** *If p and p′ are correct processes, p delivers m and p′ delivers m′, then m=m′.*

Proof. Assume by contradiction that consistency does not hold; assume correct process p delivers m, while correct process p' delivers $m' \neq m$.

Assume first *wlog* that p delivers m using the fast path. Then p must have seen m in n replicator slots. Assume now that p' also delivers m' using the fast path; then, p' must have seen m' in n replicator slots. This means that all n replicators must have changed their written

value, either from m to m', or vice-versa; this is impossible since at least $n-f$ of the replicators are correct and never change their written value (Observation 1). Process p' must have then delivered m' using the slow path instead; then, p' must have seen signed copies of m' in $n-f$ replicator slots. This means that $n-f$ replicators, including at least one correct replicator, must have changed their value from m to m', or vice-versa; this is impossible by Observation 1.

So it must be that p and p' deliver m and m', respectively, using the slow path. In this case, p sees signed copies of m in $n-f$ slots, while p' sees signed copies of m' in $n-f$ slots. Lemma B.1.2 therefore applies: p must also see a signed copy of m' or p' must also see a signed copy of m. Given there exists another validly signed value, the check at line 27 fails for p or p'. We have reached a contradiction: p does not deliver m or p' does not deliver m'. □

Lemma B.1.6 (Integrity)**.** *If some correct process delivers m and s is correct, then s previously broadcast m.*

Proof. Let a correct receiver p deliver a value, say $m \neq \perp$. To deliver, m must either be (a) the value p reads from the slots of all replicators (line 25) or (b) the signed value p reads from the slots of at least $n-f$ replicators (line 28). In both cases, for the delivery of m to occur, at least one correct replicator r contributes by writing value m (unsigned in case (a) or signed in case (b)) to its slot. Given r is a correct process, it must have copied the value it read from the sender's slot. Furthermore, a correct sender never writes any value unless it *cb-broadcasts* it. Therefore, m must have been broadcast by the sender s. □

Execution. We provide an example of an execution breaking consistency when the collect operation is used instead of the scan operation in Algorithm 4.1. Let there be a Byzantine sender s and $n = 3$ replicators. Let p_1, p_2 be two correct receivers, r_1, r_2 two correct replicators and let replicator r_3 be Byzantine. Initially, let s write m_1 signed in its slot. Let receiver p_2 start its collect. It reads the slot of r_1 which it finds empty, and sleeps. Let r_1, r_3 copy m_1 signed in their slot, while r_2 sleeps. Let p_1 perform its collect, find two signed copies of m_1 and deliver m_1 via the check at line 28. Let s change its value to m_2 signed, while r_3, being Byzantine, changes its value to m_2 signed. We resume r_2 and let it copy m_2 signed. We resume p_2's collect, continuing to read r_2, r_3 slots, seeing two values of m_2 signed (recall it previously read r_1's slot while it was empty) and delivering m_2 via the check at line 28.

B.2 Correctness of Reliable Broadcast

Invariant B.2.1. *Let S and S' be two valid ReadySets for m and m', respectively. Then, $m = m'$.*

Proof. By contradiction. Assume there exist valid *ReadySets* S and S' for different values $m \neq m'$. Set S (resp. S') consists of at least $n-f$ signed m (resp. signed m') messages. Then there exist correct replicators r and r' such that r writes m and its signature to its *Echo* slot

and r' writes m' and its signature to its *Echo* slot. This is impossible since correct replicators only write $*$ in their *Echo* slots once they have *cb-delivered* $\langle \text{INIT}, * \rangle$ from the sender. By the consistency property of Consistent Broadcast, m must be equal to m'. □

Lemma B.2.2 (Validity)**.** *If a correct process s broadcasts m, then every correct process eventually delivers m.*

Proof. Assume the sender s is correct and broadcasts m. Let p be a correct receiver that tries to deliver s's message.

Since the sender is correct, it *cb-broadcasts* $\langle \text{INIT}, * \rangle$. By the validity property of Consistent Broadcast, all correct replicators will eventually deliver $\langle \text{INIT}, * \rangle$ from s. Then, all correct replicators will write m to their *Echo* message sub-slots, compute a signature for m and write it to their *Echo* signature sub-slots. If all replicators are correct and they copy m in a timely manner, then p is able to deliver m via the fast path (at line 43).

All correct replicators will eventually read each other's signed messages m; thus every correct replicator will be able to either (a) create a valid *ReadySet* and write it to its *Ready* slot or (b) copy a valid *ReadySet* to its *Ready* slot. Thus, p will eventually be able to read at least $n - f$ valid *ReadySets* for m and deliver m via the slow path (at line 45). □

Lemma B.2.3 (No duplication)**.** *Every correct process delivers at most one message.*

Proof. Correct processes only deliver at lines 43 or 45. Immediately after a correct p process delivers a message, p exits the *while* loop and thus will not deliver again. □

Lemma B.2.4 (Consistency)**.** *If p and p' are correct processes, p delivers m and p' delivers m', then $m{=}m'$.*

Proof. By contradiction. Let p, p' be two correct receivers. Let p deliver m and p' deliver $m' \neq m$. We consider 3 cases: (1) p and p' deliver their messages via the fast path, (2) p and p' deliver their messages via the slow path, and (3) (*wlog*) p delivers via the fast path and p' delivers via the slow path.

(1) p and p' must have delivered m and m' respectively, by reading m (resp. m') from the *Echo* slots of n replicators. Thus, there exists at least one replicator r such that p read m from r's *Echo* slot and p' read m' from r's *Echo* slot. This is impossible since correct replicators never overwrite their *Echo* slots.

(2) p and p' must have each read $n - f$ valid *ReadySets* for m and m', respectively. This is impossible by Invariant B.2.1.

(3) p' read at least one valid *ReadySet* for m'. To construct a valid *ReadySet*, one requires a signed set of $n - f$ values for m'. Thus, at least one correct replicator r must have written

m' to its *Echo* slot and appended a valid signature for m'. Process p delivered m by reading m from the *Echo* slots of all n replicators, which includes r. This is impossible since correct replicators never overwrite their *Echo* slots.

□

Lemma B.2.5 (Integrity)**.** *If some correct process delivers m and s is correct, then s previously broadcast m.*

Proof. Let p be a correct receiver that delivers m and let the sender s be correct. We consider 2 cases: (1) p delivers m via the fast path and (2) p delivers m via the slow path.

(1) Fast Path. p must have read m from the *Echo* slot of at least one correct replicator r. Replicator r writes m to its slot only upon *cb-delivering* $\langle \text{INIT}, m \rangle$ from s. By the integrity property of Consistent Broadcast, s must have broadcast m. Moreover, a correct sender only invokes *cb-broadcast*($\langle \text{INIT}, m \rangle$) upon a *rb-broadcast* event for m.

(2) Slow Path. p must have read at least one valid *ReadySet* for m. A *ReadySet* consists of a signed set of $n-f$ values for m. Thus, at least one correct replicator r must have written m signed to its *Echo* slot. The same argument as in case (1) applies.

□

Lemma B.2.6 (Totality)**.** *If some correct process delivers m, then every correct process eventually delivers a message.*

Proof. Let p be a correct receiver that delivers m. We consider 2 cases: (1) p delivers m via the fast path and (2) p delivers m via the slow path.

(1) Fast Path. p must have read m from the *Echo* slots of all n replicators, which include $n-f$ correct replicators. These $n-f$ correct replicators must eventually append their signature for m. Every correct replicator looks for signed copies of m in other replicators' *Echo* slots. Upon reading $n-f$ such values, each correct replicator is able to construct and write a valid *ReadySet* to its *Ready* slot (or copy a valid *ReadySet* to its *Ready* slot from another replicator). Thus, every correct receiver will eventually read $n-f$ valid *ReadySets* for m and deliver m via the slow path.

(2) Slow Path. p must have read valid *ReadySets* for m from the slots of $n-f$ replicators, which must include at least one correct replicator r. Since r is correct, r will never remove its *ReadySet* for m. Thus, every correct replicator will eventually either (a) copy r's *ReadySet* to their own *Ready* slots or (b) construct and write a *ReadySet* to their *Ready* slots. Note that by Invariant B.2.1, all valid *ReadySets* must be for the same value m. Thus, every correct receiver will eventually read $n-f$ valid *ReadySets* for m and deliver m via the slow path.

□

B.3 Correctness of Consensus and Additional Details

B.3.1 Valid messages

A ⟨PREPARE, *view, val, proof*⟩ message is considered valid by a (correct) process if:

- the process is part of *view*,
- the broadcaster of the PREPARE is the coordinator of *view*, i.e., *view* % n,
- when *view* = 0, *proof* = ∅ and *val* can be any value ≠ ⊥
- when *view* > 0, the estimate matches the highest view tuple in *proof* and the *proof* set is valid, i.e., it contains a set of $n - f$ non-conflicting view-change certificates for view *view*; in case all tuples in *proof* are still the init value $(0, \perp, \emptyset)$, any estimate is a valid estimate,
- the process did not previously accept a different PREPARE in *view*.

A ⟨COMMIT, *view, val*⟩ message is considered valid by a (correct) process if:

- the process is part of *view*,
- *val* can be any estimate,
- the broadcaster did not previously send a view change message for $view' > view$,
- the broadcaster did not previously send another COMMIT message for $val' \neq val$ in the same *view*.

A ⟨VIEWCHANGE, *view*+1, $(view_{val}, val, proof_{val})$⟩ message from process j is considered valid by a (correct) process if:

- $val \in (view_{val}, val, proof_{val})$ corresponds to the latest non-empty value broadcast in a ⟨COMMIT, $view_c$, val_c⟩, $val = val_c$ and $view_{val} = view_c$ ($\leq view$) and $proof_{val}$ is a valid proof for *val* (either consists of non-conflicting certificates that support *val* as highest view-tuple or all tuples are with their init value; all VIEWCHANGE and VIEWCHANGEACK messages must be for view_{val}),
- $val \in (0, val, proof_{val})$, $proof_{val}$ is ∅,
- if for each view $view' \leq view$, ⟨COMMIT, $view'$, ⊥⟩ from j are empty; then $(view_{val}, val, proof_{val})$ must be equal to $(0, \perp, \emptyset)$,

- j must have sent a single COMMIT message each view $view' \leq view$,
- j did not send another VIEWCHANGE message this view, $view$+1.

B.3.2 Agreement

Lemma B.3.1. *In any view v, no two correct processes accept* PREPARE *messages for different values $val \neq val'$.*

Proof. Let i, j be two correct processes. Any correct process accepts a PREPARE messages only from the current view's primary (line 12).

A correct primary p never broadcasts conflicting PREPARE messages (i.e., same view v, but different estimates val, val', $val \neq val'$). This means, i, j must receive the same PREPARE message. By Lemma B.3.9, both i and j consider the PREPARE message from p valid.

A faulty primary p' may broadcast conflicting PREPARE messages. Assume the primary broadcasts (k, ⟨PREPARE, v, val, $proof$⟩) and (k',⟨PREPARE, v, val', $proof'$⟩) where k, k' are the broadcast sequence numbers used. We distinguish between the following cases:

1. $k < k'$: By the FIFO property, any correct replica must process message k of p' before processing message k'. If process i accepts the k^{th} message of p', following the consensus protocol, i will not accept a second PREPARE message in the same view v, i.e. message k'. Similarly for correct replica j.
2. $k > k'$: The argument is similar to (1).
3. $k = k'$: In this case, p' equivocates. If a message gets delivered by both i and j, then the message is guaranteed to be the same by the consistency property of Consistent Broadcast.

We conclude correct replicas agree on the PREPARE message accepted within the same view. □

Lemma B.3.2. *In any view v, no two correct processes call try_decide with different values val and val'.*

Proof. By contradiction. Let i, j be two correct processes. Assume in view v, processes i, j call *try_decide* with value val, respectively val'. To call *try_decide*, the condition at line 24 must be true for both i and j. This means i, (resp. j) accepts a valid PREPARE message supporting val (resp. val') and a set of $n - f$ COMMIT messages supporting val (resp. val'). By Lemma B.3.1, correct processes cannot accept different PREPARE messages and consequently cannot call *try_decide* with different values since $aux_i = aux_j$. □

Lemma B.3.3. *Let a correct process i decide val in view v. For view v+1, no valid proof can be constructed for a different estimate $val' \neq val$.*

Proof. By contradiction. Let i decide *val* in view v. Assume the contrary and let there be a valid proof such that *(v+1, val', proof)*.

Given $v+1 > 0$, *proof* cannot be $\emptyset$. It must be the case that the proof supporting val' consists of a set of $n-f$ non-conflicting view-change certificates. Each view-change certificate consists of a VIEWCHANGE message with format ⟨VIEWCHANGE, v+1, *(view, value, $proof_{val}$)*⟩ and f corresponding VIEWCHANGEACK messages. Any view-change certificate requires the involvement of at least one correct replica, namely, either a correct replica is the broadcaster of a VIEWCHANGE message or a correct replica validates a VIEWCHANGE message, by sending a corresponding VIEWCHANGEACK.

For val' to be consistent with *proof*, *proof* must contain either (a) at least one view-change certificate with tuple *(v, val', $proof_{val'}$)* and no other view-change certificate s.t. its tuple has a different value for the same view, v, i.e., $\nexists$ *(v, val, $proof_{val}$)*, with v the highest view among the $n-f$ tuples or (b) only view-change certificates with tuples having the initial value $(0, \perp, \emptyset)$ so that any value is a valid value. Let R_1 denote the set of processes that contributed with a VIEWCHANGE message, which is then part of a view-change certificate in *proof*.

Given i decided *val* in view v, i received $n-f$ COMMIT messages for *val* (line 24). Such processes must have received a valid PREPARE message and updated their view-change tuple together with their auxiliary in lines 14 and 15, before sending a COMMIT message. Let R_2 denote the set of processes that contributed with a COMMIT message for *val*.

These two sets, R_1 and R_2, must intersect in at least one replica j. Replica j must have used Consistent Broadcast for its view-change message: $(k_{vc}$, ⟨VIEWCHANGE, v+1, *(v, val', $proof_{val'}$)*⟩); the argument is similar for the case $(k_{vc}$, ⟨VIEWCHANGE, v+1, $(0, \perp, \emptyset)$⟩, where k_{vc} is the broadcast sequence number used; otherwise it could have not gathered enough VIEWCHANGEACKs, since correct replicas do not accept messages not delivered via the broadcast primitive. Similarly, j must have used Consistent Broadcast for its COMMIT message: $(k_c$, ⟨COMMIT, v, val⟩), where k_c is the broadcast sequence number used; otherwise i would not have accepted the COMMIT message.

If j is correct, and sends a COMMIT message for *val*, it broadcasts a VIEWCHANGE message with its true estimate, *val*. Hence, the R_1 set of non-conflicting view-change messages must contain a tuple *(v, val, $proof_{val}$)*. This yields either a set of conflicting view-change certificates if ∃ another view-change certificate for *(v, val', $proof_{val'}$)*, or a conflict between *proof* and val' as matching estimate (since v is the highest-view and the value associated with this tuple corresponds to estimate *val* and not val').

If j is Byzantine, we distinguish between the following cases:

1. $k_c < k_{vc}$ (j broadcasts its COMMIT message before it broadcasts its VIEWCHANGE message). In this case, no correct process sends a VIEWCHANGEACK for j's VIEWCHANGE message. By the FIFO property, a correct process first delivers the k_c message and then k_{vc} message. In order to validate a VIEWCHANGE message, the last non-empty value broadcast in a COMMIT must correspond to the value broadcast in the VIEWCHANGE. Since these do not match, no correct process sends a VIEWCHANGEACK for j's VIEWCHANGE. Hence, the VIEWCHANGE message of j does not gather sufficient ACKs to form a view-change certificate and be included in *proof*.
   Note: If j were to broadcast two COMMIT messages in view v, one supporting *val* and another supporting *val'* before broadcasting its VIEWCHANGE message supporting *val'*, no correct process ACKs its VIEWCHANGE message since j behaves in a Byzantine manner, i.e., no correct process broadcasts two (different) COMMIT messages within the same view.

2. $k_{vc} < k_c$ (j broadcasts its VIEWCHANGE message before it broadcasts its COMMIT message). In this case, process i must have first delivered the VIEWCHANGE message from j. Consequently, i does not accept j's COMMIT message as valid. This contradicts our assumption that i used this COMMIT message to decide *val*.

3. $k_{vc} = k_c$ (j equivocates). By the properties of Consistent Broadcast, correct processes either deliver j's COMMIT message, case in which the VIEWCHANGE message does not get delivered by any correct replica, and consequently does not gather sufficient VIEWCHANGEACK to form a view-change certificate (for neither *val'* nor $\perp$); or correct processes deliver j's VIEWCHANGE message, case in which the COMMIT message does not belong to R_2, i does not decide.

We conclude, if i decided *val* in view v, no valid proof can be constructed for view *v+1* and *val'* $\neq$ *val*. □

Lemma B.3.4. *Let a correct process i decide val in view v. For any subsequent view $v' > v$, no valid proof can be constructed for a different estimate val' $\neq$ val.*

Proof. We distinguish between the following two cases: (1) $v' = v+1$ and (2) $v' > v$.

Case 1: Follows from Lemma B.3.3.

Case 2: By contradiction. Let process i decide *val* in view v. Assume the contrary and let $v' > v$ be the lowest view in which there exists a valid proof for *val' $\neq$ val*, i.e., *(v',val', proof)*.

A valid proof supporting *val'* must contain $n - f$ non-conflicting view-change certificates out of which (a) one view-change certificate supports *val'* or (b) all view-change certificates claim $\perp$. A view-change certificate consists of a VIEWCHANGE message and f VIEWCHANGEACK messages. This means, at least one correct process must validate a VIEWCHANGE message by broadcasting a VIEWCHANGEACK message, or be the producer of a VIEWCHANGE message.

(a) For val' to be the representative value of the $n-f$ view-change certificates in *proof*, one of the view-change messages must contain a tuple with the highest view among all $n-f$ tuples. Let this tuple be $(v'-1, val', proof_{val'})$ such that $v'-1$ is the highest view possible before entering v'. This tuple must then come from view $v'-1$ with a valid proof, $proof_{val'}$ supporting the fact that val' is a valid value.

By assumption, the only valid proof that can be constructed in views prior to v' but succeeding v is for estimate *val*. Hence, there is no valid proof, $proof_{val'}$ for val' in view $v'-1$. In the case in which the producer of the VIEWCHANGE is correct, it will not construct a VIEWCHANGE message with an invalid proof. The vc_i variable is only updated if the PREPARE message is valid. In the case in which the producer of the VIEWCHANGE is faulty, it will not gather the necessary VIEWCHANGEACK to form a view-change certificate given correct replicas do not validate a VIEWCHANGE message with an invalid proof or in which the estimate value contradicts the proof. This contradicts our initial assumption that there exists a valid view-change certificate supporting val'.

(b) All VIEWCHANGE messages in *proof* have tuples $(0, \perp, \emptyset)$ so that any estimate value is valid value. Let this set be denoted by R_1. Since i decided *val* in view v, a set of R_2 replicas contributed with a COMMIT value for *val*. Sets R_1 and R_2 must intersect in one replica, say j. If j sends COMMIT messages in subsequent views for a value $\perp$, j must send a VIEWCHANGE message matching its latest non-empty COMMIT message, i.e., $\langle$VIEWCHANGE, v', $(v, val, proof_{val})\rangle$, in order to gather sufficient VIEWCHANGEACK and hence form a view-change certificate. If j sends a COMMIT message in any subsequent view for a value $val' \neq \perp$, the only possible valid proof is for value *val*, see case (a). Whichever the case, at least one view-change certificate in *proof* must contain a view-change message with a non-empty tuple which contradicts our assumption that all view-change certificates are for $\perp$. □

Theorem B.3.5 (Agreement)**.** *If correct processes i and j decide val and val', respectively, then $val = val'$.*

Proof. We distinguish two cases: (1) decision in the same view (2) decision in different views.

Case 1: decision in the same view. Follows from Lemma B.3.2.

Case 2: decision in different views. By contradiction. Let i, j be two correct processes. Assume processes i and j decide two different values, *val*, respectively val', in views v, respectively v'. Let $v < v'$ *wlog*.

To decide, a correct process must receive a valid PREPARE message and $n-f$ COMMIT messages for the same estimate, line 24. When i decides *val* in view v, by Lemma B.3.4, from view *v+1* onward, the only valid *proof* supports estimate *val*. Hence, a valid PREPARE message can only contain an estimate for *val* and at any view-change procedure, no VIEWCHANGE supporting val' is able to form a view-change certificate. Given process j only accepts valid PREPARE messages, j cannot adopt val' as its auxiliary, aux_j. This means j cannot decide val'. Given

process j only collects a set of (non-conflicting) view-change certificates, j cannot adopt val' as its estimate est_j. □

B.3.3 Integrity

Theorem B.3.6 (Integrity)**.** *No correct process decides twice.*

Proof. A correct process may call *try_decide* (line 25) multiple times. Yet, once a correct process calls *decide* (line 42), the *decided* variable is set to *true* and hence the if statement is never entered again. □

B.3.4 Validity

Theorem B.3.7 (Weak validity)**.** *If all processes are correct and some process decides val, then val is the input of some process.*

Proof. Assume a correct process decides *val*. Following the steps in the algorithm, a correct process only decides a value for which it receives a valid PREPARE message and $n - f$ COMMIT messages, in the same view (line 24). It is either the case the value in the PREPARE message comes from the previous view or it is the input value of the current view's primary (line 10). For the latter, validity is satisfied. For the former, the value in the previous view must come from one of the VIEWCHANGE messages. Which is either an input value of a prior view's primary or the value of a previous view message. We continue by applying the same argument inductively, backward in the sequence of views, until we reach a view in which the value was the input value of a primary. This shows that *val* was proposed by the primary in some view. □

B.3.5 Termination

Lemma B.3.8. *Two correct processes cannot send conflicting* VIEWCHANGE *messages.*

Proof. Assume the contrary and let v be the earliest view in which correct processes i and j send conflicting VIEWCHANGE messages m_1 and m_2, respectively. Let $vc_i = (view_i, val_i, proof_i)$ and $vc_j = (view_j, val_j, proof_j)$ be the view-change tuples in m_1 and m_2, respectively. Since m_1 and m_2 conflict, it must be the case that $view_i = view_j$ and $\perp \neq val_i \neq val_j \neq \perp$. Thus, in view $view_i = view_j$, i and j must have received and accepted PREPARE messages for different values val_i and val_j. This contradicts Lemma B.3.1. □

Lemma B.3.9. *A* PREPARE, COMMIT *or* VIEWCHANGE *message from a correct process is considered valid by any correct process.*

Proof. A correct process i only sends a PREPARE message if it is the coordinator of that view

(line 9). When $view = 0$, est_i is initialized to $\perp$ which leads i to set $init_i$ to v_i (line 10). The PREPARE message has the following format: $\langle$PREPARE, 0, v_i, $\emptyset\rangle$ which matches the required specification for a valid PREPARE. When $view > 0$, any correct process updates its $proof_i$ and est_i before increasing its $view_i$ variable, i.e. moving to the next view. A correct process would update these two vars according to the protocol, lines 30 and 31. As before, in case $est_i = \perp$, i to set $init_i$ to v_i, otherwise it carries est_i (line 10). The PREPARE message has the following format: $\langle$PREPARE, $view_i$, $init_i$, $proof_i\rangle$ which matches the required specification for a valid PREPARE.

A correct process i broadcasts exactly one COMMIT message in *view* (line 20) after it either (a) hears from the coordinator of the current view or (b) starts suspecting the coordinator. In case (a) i's message contains the estimate of the coordinator (line 14), while in case (b) it contains $\perp$ (line 17). In any of the two cases, i's VIEWCHANGE message strictly follows the COMMIT message (lines 28 and 20). The behaviour is in-line with the specification.

A correct process i broadcasts exactly a single VIEWCHANGE message in one *view* (line 28) with its vc_i. Process i update its view-change tuple, vc_i, only when it receives a valid PREPARE message. Such message is ensured to be in accordance with the prior specifications for a valid PREPARE message. Notice that a valid PREPARE message cannot be $\perp$, and hence vc_i is either its initial value, $(0, \perp, \emptyset)$ or a valid tuple *(view, val, proof)*. The data in vc_i is updated at the same time aux_i is updated, upon receiving a valid PREPARE, and these two variables indicate the same estimate (lines 14 and 15). The aux_i is then send via a COMMIT message within the same view (line 20). This ensures that the broadcast of i's latest non-empty COMMIT corresponds to the data in its vc_i variable. □

Let i be a correct process. For a given execution E, we denote by $\mathcal{V}(i)$ the set of views in which i enters. We denote $v_{max}(i) = \max \mathcal{V}(i)$; by convention $v_{max}(i) = \infty$ if $\mathcal{V}(i)$ is unbounded from above.

Lemma B.3.10. *For every correct process i, $v_{max}(i) = \infty$*

Proof. Assume the contrary and let *wlog* i be the process with the lowest v_{max}. Since i never progresses past view $v_{max}(i)$, i must be blocked forever in one of the *wait until* statements at lines 12, 22, or 29. We now examine each such case:

1. Line 12. If the primary p of view $v_{max}(i)$ is faulty and does not broadcast a valid PREPARE message, then eventually i times out on the primary and progresses past the *wait until* statement. If p is correct, then p eventually reaches view $v_{max}(i)$ and broadcasts a PREPARE message m. By the validity property of Consistent Broadcast, i eventually delivers m from p. By Lemma B.3.9, i considers m valid and thus progresses past the *wait until* statement.
2. Line 22. By our choice of i, every correct process must eventually reach view $v_{max}(i)$. Given the argument at item (1) above, no correct process can remain blocked forever

at the *wait until* statement in line 12, thus every correct process eventually broadcasts a COMMIT message in view $v_{max}(i)$. By the validity property of Consistent Broadcast and by Lemma B.3.9, i eventually delivers all such messages and considers them valid. Therefore, i must eventually deliver valid PREPARE messages from $n-f$ processes and progress past the *wait until* statement.

3. Line 29. By our choice of i, every correct process must eventually reach view $v_{max}(i)$. Given the argument at items (1) and (2) above, no correct process can remain blocked forever at the *wait until* statements in lines 12 and 22, thus every correct process eventually broadcasts a VIEWCHANGE message in view $v_{max}(i)$. By the validity property of Consistent Broadcast and by Lemma B.3.9, every correct process eventually delivers all such VIEWCHANGE messages and considers them valid. Thus, for every VIEWCHANGE message m sent by a correct process in view $v_{max}(i)$, every correct process eventually broadcasts a VIEWCHANGEACK message $mAck$ with m's digest; furthermore, i receives and considers valid each such $mAck$. Thus, i eventually gathers a set of $n-f$ view-change certificates in view $v_{max}(i)$, which are non-conflicting by Lemma B.3.8. This means that i is eventually able to progress past the *wait until* statement.

We have shown that i cannot remain blocked forever in view $v_{max}(i)$ in any of the *wait until* statements. Thus, i must eventually reach line 32 and increase $view_i$ to $v_{max}(i)+1$. We have reached a contradiction. □

We define a view v to be *stable* if in v: (1) the coordinator is correct and (2) no correct process times out on another correct process.

Theorem B.3.11 (Termination)**.** *Eventually every correct process decides.*

Proof. We will show that every correct process eventually calls *try_decide*, which is sufficient to prove the result. By our assumption of eventual synchrony, there is a time T after which the system is synchronous. We can also assume that after T, no correct process times out on another process. Let i be a correct process. Let v^* be the earliest view such that: (1) i enters v^* after time T and (2) the primary of v^* is correct. Recall that by Lemma B.3.10, i and all other correct processes are guaranteed to eventually reach view v^*. Let p be the (correct) primary of v^*. By our choice of v^*, p broadcasts a PREPARE message m in v^*, which is received and considered valid by all correct processes (by the validity property of Consistent Broadcast and Lemma B.3.9). Thus all correct processes will set their *aux* variable to the value *val* contained in m, and broadcast a COMMIT message with *val*. Process i must eventually deliver these COMMIT messages and consider them valid, thus setting at least $n-f$ entries of R_i to *val* in line 23. Therefore, the check at line 24 will succeed for i and i will call *try_decide* at line 25. □

C Appendix for uBFT

C.1 Correctness of Consistent Tail Broadcast

In this section, we provide a correctness argument for the implementation of CTBcast given in Algorithm 5.1.

Observation 2. *A correct broadcaster p TBcast-broadcasts at most one* `LOCK` *message and at most one* `SIGNED` *message per sequence number k. Moreover, both of these broadcasts hold the same message m.*

Observation 3. *A correct process p TBcast-broadcasts at most one* `LOCKED` *message per sequence number k.*

Proof. Correct processes only broadcast `LOCKED` messages at line 16. Moreover, $\texttt{locks}[k\%t]$, which is only modified at lines 15 and 29, is updated strictly monotonically. Thus, once the branch is entered at line 14 (and thus $\texttt{locks}[k\%t]$ updated at line 15), it cannot be entered for the same k, which ensures that line 16 is executed at most once per k at correct processes. □

Lemma C.1.1 (Tail-Validity)**.** *If a correct process p broadcasts* (k, m) *and never broadcasts a message* (k', m') *with* $k' \geq k + t$*, then all correct processes eventually deliver* (k, m).

Proof. Let p, k, m be as in the statement of the lemma and let q be a correct receiver. We will show that q eventually delivers (k, m), which is sufficient to prove the lemma.

Since p is correct, p TBcast-broadcasts $\langle \texttt{SIGNED}, k, m, sig \rangle$ with a valid signature. Since both p and q are correct, q eventually TBcast-delivers it.

The $\langle \texttt{SIGNED}, k, m, sig \rangle$ message of p will trigger at q the event at line 25. Given that p's signature is valid, the check at line 26 succeeds. By the premise, p does not broadcast any message with sequence number $k' \geq k + t$, so $\texttt{locks}[k\%t]$, which is only modified at lines 15 and 29, cannot contain a value greater than k. Moreover, since no other message m is broadcast for k (Observation 2), if $\texttt{locks}[k\%t]$ already contains k, it must also contain the message m.

Thus, q enters the if branch at line 28. Since p is correct and no process can forge p's signature, no validly signed entry (k, m') with $m \neq m'$ can exist in any process's SWMR slot, so q does not return by triggering the check at line 33. Finally, since p does not broadcast any message with sequence number $k' \geq k + t$, no process's SWMR slot can contain a validly signed entry $(k', \cdot)$ with $k' > k$ and $k' \equiv k \pmod{t}$, so q does not return by triggering the check at line 35. Therefore, q must call `deliver_once`(k, m) at line 37. If this call does not deliver (k, m), it must have been delivered before. Thus, q eventually delivers (k, m). □

Lemma C.1.2 (Agreement)**.** *If p and q are correct processes, p delivers (k, m) from r, and q delivers (k, m') from r, then $m = m'$.*

Proof. Assume towards a contradiction that $m \neq m'$. We consider two cases: (1) at least one process delivers via the fast path, and (2) both processes deliver via the slow path.

In case (1), assume *wlog* that p delivers via the fast path. Then p must have TBcast-delivered a `LOCKED` message from q for (k, m). So q must have TBcast-broadcast a `LOCKED` message at line 16. By Observation 3, q cannot have broadcast a `LOCKED` message for (k, m'). Thus, it cannot have delivered m' via the fast path. Moreover, q must have put m in `locks`$[k\%t]$ at line 15. Thus q cannot enter the if branch at line 28 and cannot deliver (k, m') via the slow path either, hence a contradiction.

In case (2), assume *wlog* that p writes (k, sig, m) to `SWMR`$[p][k\%t]$ (line 30) before q writes (k, sig', m') to `SWMR`$[q][k\%t]$. Thus, when q reads p's $k\%t$ slot at line 31, q sees either (i) (k, sig, m) or (ii) $(k'', \cdot, \cdot)$ with $k'' > k$, and $k'' \equiv k \pmod{t}$. In case (i), q will return by triggering the check at line 33, and thus not deliver (k, m'), a contradiction. In case (ii), q will return by triggering the check at line 35, and thus not deliver, a contradiction. □

Lemma C.1.3 (Integrity)**.** *If a correct process delivers (k, m) from p and p is correct, p must have broadcast (k, m).*

Proof. Let p and q be correct processes and assume q delivers (k, m) from p. There are two possible cases: (1) q delivers using the fast path at line 23, or (2) q delivers using the slow path at line 37.

In case (1), q must have TBcast-delivered `LOCKED` messages for (k, m) from all processes, including itself. Therefore q must have TBcast-broadcast a `LOCKED` message for (k, m) at line 16 after TBcast-delivering a `LOCK` message for (k, m) from p. Thus, p must have TBcast-broadcast a `LOCK` message, which p can only do as part of the CTBcast-broadcast call. So p must have broadcast (k, m).

In case (2), q must have TBcast-delivered a valid `SIGNED` message from p for (k, m). Since p is correct and no process can forge its signature, p must have broadcast a `SIGNED` message for (k, m). So p must have broadcast (k, m). □

Lemma C.1.4 (No duplication)**.** *No correct process delivers* $(k, *)$ *from p twice.*

Proof. Correct processes only deliver through `deliver_once`. Lines 40 and 41 ensure that a correct process only triggers $deliver$ at most once per sequence number k. □

Theorem C.1.5. *From Lemmas C.1.1, C.1.2, C.1.3 and C.1.4, Algorithm 5.1 implements all the properties of Consistent Tail Broadcast.* □

C.2 Correctness of Consensus

Algorithm C.1: Common Case (stable leader)

```
1  CTBcasts FIFO-deliver and block upon a Byzantine message

3  upon Init:
4    view = 0
5    next_slot = 0
6    checkpoint = (app_state: Initial, open_slots: [0, 99])_Σ
7    for each replica:
8      state[replica] = {
9        view = 0, seal_view = ⊥, new_view = ⊥,
10       prepares: Map⟨slot, PREPARE⟩ = {},
11       commits: Map⟨slot, COMMIT⟩ = {},
12       checkpoint = (Initial, [0, 99])_Σ }

14 def Propose(req):
15   wait (leader(view) == me and next_slot in checkpoint.open_slots and NEW_VIEW
       ↪ broadcast if view > 0)
16   CTBcast-bcast ⟨PREPARE, view, next_slot++, req⟩

18 upon CTBcast-deliver ⟨PREPARE, v, s, r⟩ from p as P:
19   state[p].prepares[s] = P
20   if v != view or s ∉ checkpoint.open_slots: return
21   TBcast-bcast ⟨WILL_CERTIFY, v, s⟩ # Fast path
22   TBcast-bcast ⟨CERTIFY, sign(P)⟩ # Slow path

24 # Fast path
25 upon TBcast-deliver ⟨WILL_CERTIFY, v, s⟩ from 2f+1:
26   if v != view or s ∉ checkpoint.open_slots: return
27   TBcast-bcast ⟨WILL_COMMIT, v, s⟩

29 upon TBcast-deliver ⟨WILL_COMMIT, v, s⟩ from 2f+1:
30   if v != view or s ∉ checkpoint.open_slots: return
31   trigger once Decide(s, state[leader(v)].prepares[s].req)

33 # Slow path
34 upon TBcast-deliver ⟨CERTIFY, ⟨P., v, s, _⟩_σ⟩ from f+1 as P_Σ:
35   if v != view or s ∉ checkpoint.open_slots: return
36   CTBcast-bcast ⟨COMMIT, P_Σ⟩
```

```
38 upon CTBcast-deliver ⟨COMMIT, P_Σ⟩ from p as C:
39   state[p].commits[P_Σ.slot] = C
40   if delivered f+1 COMMIT with a matching PREPARE:
41     trigger once Decide(P_Σ.slot, P_Σ.req)

43 # Checkpoints
44 after having decided on all checkpoint.open_slots:
45   wait for all decided requests to be applied to the App
46   next_cp = (App.Snapshot(), checkpoint.open_slots + 100)
47   TBcast-bcast ⟨CERTIFY_CHECKPOINT, sign(next_cp)⟩

49 upon TBcast-deliver ⟨CERTIFY_CHECKPOINT, c_σ⟩ from f+1 as C_Σ:
50   MaybeCheckpoint(C_Σ)

52 upon CTBcast-deliver ⟨CHECKPOINT, C_Σ⟩ from p as CP:
53   state[p].checkpoint = CP
54   forget state[p].commits and prepares ∉ C_Σ.open_slots
55   MaybeCheckpoint(C_Σ)

57 def MaybeCheckpoint(C_Σ):
58   if C_Σ supersedes checkpoint:
59     checkpoint = C_Σ
60     App.BringUpToSpeed(checkpoint)
61     TBcast-bcast ⟨CHECKPOINT, checkpoint⟩
```

Algorithm C.2: View Change

```
1  upon suspicion of leader(view): ChangeView()

3  def ChangeView():
4    for each ⟨WILL_COMMIT, v|_{v==view}, s⟩ bcast via TBcast:
5      wait to have broadcast a matching COMMIT or CHECKPOINT
6    CTBcast-bcast ⟨SEAL_VIEW, ++view⟩

8  upon CTBcast-deliver ⟨SEAL_VIEW, v⟩ from p as SV:
9    state[p].seal_view = SV
10   state[p].view = v
11   send_{leader(v)} ⟨CRTFY_VC, v, sign((p, state[p]\new_view))⟩

13 upon deliver f+1 matching ⟨CRTFY_VC, v|_{v==view}, s_σ⟩ about f+1 replicas as C:
14   if me != leader(view): return
15   CTBcast-bcast ⟨NEW_VIEW, C⟩
16   MaybeCheckpoint(highest checkpoint in C)
17   for s in checkpoint.open_slots:
18     CTB-bcast ⟨PREPARE, v, s, MustPropose(s, C)⟩
19   next_slot = checkpoint.open_slots.last + 1

21 upon CTBcast-deliver ⟨NEW_VIEW, certificates⟩ from p as NV:
22   state[p].new_view = NV
23   while view != NV.view + 1: ChangeView()

25 def MustPropose(slot, certificates):
26   if slot > max open slot in certificates: return Any
27   return latest committed req for slot in certificates or ⊥
```

Algorithm C.3: CTBcast Summaries

```
1  after CTBcast-deliver the message with id % tail == 0 from p:
2    send_p ⟨CERTIFY_SUMMARY, sign((p, id, state[p]))⟩

4  every tail invocations of CTBcast-bcast:
5    block calls to CTBcast-bcast

7  upon deliver ⟨CERTIFY_SUMMARY, (me, id, _)_σ⟩ from f+1 as S_Σ:
8    TBcast-bcast ⟨SUMMARY, S_Σ⟩
9    unblock calls to CTBcast-bcast

11 upon TBcast-deliver ⟨SUMMARY, (p, id, history)_Σ⟩:
12   when a gap is detected in the delivery of CTBcast from p:
13     if the latest message delivered from p is lower than id:
14       deliver in order p's missed CTBcast messages in history without running the
           ↪ Byzantine checks (Alg. C.4)
15       continue delivering p's CTBcast messages after id
```

Algorithm C.4: CTBcast's Byzantine Checks

```
1  def valid ⟨PREPARE, v, s, r⟩ from p:
2    state[p].view == v and leader(v) == p and
3      s in state[p].checkpoint.open_slots and
4      p never prepared slot s before in v and
5      (v == 0 or (state[p].new_view != ⊥ and
6        r == MustPropose(s, state[p].new_view)))

8  def valid ⟨COMMIT, P_Σ⟩ from p as C:
9    P_Σ.slot in state[p].checkpoint.open_slots and
10     P_Σ.view == state[p].view and
11     state[p].commits[P_Σ.slot] != C

13 def valid ⟨CHECKPOINT, C_Σ⟩ from p:
14   C_Σ supersedes state[p].checkpoint

16 def valid ⟨SEAL_VIEW, v⟩ from p:
17   state[p].view < v

19 def valid ⟨NEW_VIEW, certificates⟩ from p:
20   leader(state[p].view) == p and
21     it is p's first non-CHECKPOINT message in this view and
22     each certificate is about a different replica and
23     each certificate is signed by f+1 different replicas and
24     each certificate is about view state[p].view
```

This section gives the pseudocode of uBFT's consensus alongside a correctness argument. Algorithm C.1 describes uBFT's operation under a stable leader. Algorithm C.2 describes view changes. Algorithm C.3 describes how summaries let uBFT handle the gaps caused by CTBcast's tail-validity. Finally, Algorithm C.4 gives the explicit requirements for messages to pass CTBcast's Byzantine checks.

C.2.1 Validity

Lemma C.2.1. *For a fixed slot s, with no faulty processes, if some process p delivers and accepts $\langle \mathit{PREPARE}, v, s, r\rangle$ in Algorithm C.1 at line 18, then r must have been proposed by some correct process.*

Proof Sketch. We will prove the lemma by induction on the view v in which p accepts the PREPARE message. The base case is $v = 0$. Process p must have delivered a PREPARE message for r from the leader ℓ_0 of view 0. Since ℓ_0 is correct, it only sends PREPARE messages for values that are in its input, or for values that are part of a valid view change certificate from the previous view. Since there is no previous view in view 0, it must be that r was ℓ_0's input.

Now, for the induction step, assume that the lemma is true up to view v, and examine the case in which p accepts $\langle \texttt{PREPARE}, v+1, s, r\rangle$ in view $v+1$. All processes are assumed to be correct, so the PREPARE message must have been sent by ℓ_{v+1}, the leader of view $v+1$. Correct processes only send one PREPARE message per slot per view, so ℓ_{v+1} must have sent $\langle \texttt{PREPARE}, v+1, s, r\rangle$ either as a new proposal, or during the view change from v to $v+1$, in Algorithm C.2 at line 18. In the first case, r is by definition proposed by a correct process as part of ℓ_{v+1}'s input. In the second case, r must be a valid value (i.e., be returned by `MustPropose`), given the view change certificates for view v. There are two cases in which r is such a valid value: (1) one of the certificates contains a COMMIT messages for r in v' with $v' \leq v$, or (2) none of the certificates contain a COMMIT message for r, and r is the input of ℓ_{v+1}. In case (1), a quorum of processes must have delivered and accepted $\langle \texttt{PREPARE}, v', s, r\rangle$ messages in view v' with $v' \leq v$ and thus, by induction, r must have been proposed by some correct process. In case (2), r is also proposed by a correct process. This concludes the induction step and the proof. □

Theorem C.2.2 (Weak Validity)**.** *For a fixed slot s, with no faulty processes, if some process p decides value r in s, then r must have been proposed by some correct process.*

Proof Sketch. Process p may decide r either at (1) line 31 (fast path), or (2) at line 41 (slow path) of Algorithm C.1. Let v be the view in which p decides r. In case (1), p must have delivered and accepted a $\langle \texttt{PREPARE}, v, s, r\rangle$ in view v, so by Lemma C.2.1, r must have been proposed by some correct process. In case (2), p must have received valid COMMIT messages for r from a quorum. Thus, a quorum of processes must have delivered and accepted a PREPARE message for r, so by Lemma C.2.1, r must have been proposed by some correct process. □

C.2.2 Agreement

Observation 4. *For a fixed slot s and view v, two correct processes never deliver and accept conflicting PREPARE messages.*

Proof Sketch. Correct processes deliver and accept PREPARE messages only when coming from the leader. Furthermore, they deliver and accept at most one PREPARE message per view per slot. Thus, by the Agreement property of CTBcast, if two correct processes deliver and accept PREPARE messages in the same view, then those messages are for the same value and thus do not conflict. □

Corollary C.2.3. *For a fixed slot s and view v, two processes never broadcast conflicting valid COMMIT messages.*

Proof Sketch. Assume towards a contradiction that two processes q and p broadcast conflicting valid COMMIT messages. Given that each valid COMMIT message is made of a quorum of valid CERTIFY messages, p and q must have delivered two quorums of valid CERTIFY messages about different PREPARE messages. By definition, each quorum must contain one correct process. Moreover, a correct process only broadcasts a CERTIFY message about the PREPARE message it delivered. Thus, two correct processes delivered different PREPARE messages for the same slot in the same view. This contradicts Observation 4. □

Corollary C.2.4. *For a fixed slot s, the view change certificates corresponding to two processes cannot have conflicting COMMIT messages from the same view.*

Proof Sketch. Assume not. Then there exist processes p_1 and p_2 such that their view change certificates at the end of view v are conflicting: they contain different COMMIT messages for values r_1 and r_2, respectively, from the same view. Since each certificate contains an approval from a quorum, each certificate must have been approved by at least one correct process. Thus, a correct process must have received a COMMIT from p_1 for r_1 and, in the same view, a correct process (not necessarily the same) must have received a COMMIT from p_2 for r_2. By the Integrity property of CTBcast, this implies that p_1 and p_2 must have sent conflicting commits for the same slot and view, which is impossible by Corollary C.2.3. □

Lemma C.2.5. *For a fixed slot s and view v, if a quorum broadcasts COMMIT messages for the same value r, then no correct process accepts a PREPARE message for any other value $r' \neq r$ in any view $v' \geq v$.*

Proof Sketch. We proceed by induction on v'. The base case is $v' = v$. Since a quorum broadcasts COMMIT messages for r in view v, at least one correct process p must have broadcast a COMMIT for r. Thus, some correct process must have delivered and accepted a PREPARE message for r. Thus, by Observation 4, no correct process may accept a PREPARE for a different value $r' \neq r$ in the same view.

Now, for the induction step, assume the lemma is true up to view v', and assume that in view $v'+1$, some correct process p accepts a PREPARE message for some other value $r' \neq r$. For this to happen, r' must be a valid value according to the view change certificates provided by the leader $\ell_{v'+1}$ of view $v'+1$. Thus, at least one process q must have sent a COMMIT message

C for r' in a view $v'' \leq v'$. Furthermore, C must have been accepted by at least one correct process w, in order for q's state to have been certified by a quorum. In order for w to accept C, C's corresponding PREPARE message must have been certified, and thus accepted, by at least one correct process in view v''. This contradicts our induction hypothesis. So it is impossible for any correct process to accept a PREPARE message for r' in view $v'+1$. This completes the induction step and the proof. □

Theorem C.2.6 (Agreement)**.** *For a given slot s, correct processes cannot decide different values.*

Proof Sketch. Assume by contradiction that there exist two correct processes p_1 and p_2, such that p_1 decides r_1 in view v_1 and p_2 decides $r_2 \neq r_1$ in view v_2. Assume further *wlog* that $v_1 \leq v_2$. We consider four cases, based on whether p_1 and p_2 decide on the fast path or the slow path.

Case 1: Fast-fast. Both p_1 and p_2 decide their respective values on the fast path. If $v_1 = v_2$, then p_1 and p_2 must have accepted conflicting PREPAREs in the same view, which is impossible by Observation 4. Otherwise, if $v_1 < v_2$, then at least $f+1$ correct processes (a quorum) must have broadcast COMMIT messages for r_1 before sealing view v_1. Thus, by Lemma C.2.5, no correct process can accept a PREPARE for r_2 in v_2, so p_2 cannot decide r_2 on the fast path in v_2.

Case 2: Fast-slow. p_1 decides on the fast path and p_2 decides on the slow path. Then, p_1 must have accepted a PREPARE for r_1 in view v_1 (call this Fact 1). Moreover, p_2 must have accepted COMMIT messages for r_2 from a quorum. This implies that a quorum broadcast COMMIT messages for r_2 in some view $v_2' \leq v_2$ (call this Fact 2). If $v_2' \leq v_1$, then we reach a contradiction with Fact 1 by Lemma C.2.5. If $v_2' > v_1$, then a quorum of correct processes must have broadcast COMMIT messages for r_1 before sealing v_1; thus, by Lemma C.2.5, we reach a contradiction with Fact 2, since no correct process could have accepted a PREPARE for r_2 in v_2.

Case 3: Slow-fast. This case is symmetric with Case 2 above.

Case 4: Slow-slow. If both p_1 and p_2 decide on the slow path, then both processes must have accepted COMMIT messages from a quorum. Let v_1' and v_2' be the views in which the COMMIT messages accepted by p_1 and p_2, respectively, were sent. Assume *wlog* that $v_1' \leq v_2'$. Then, by Lemma C.2.5, no correct process could have accepted a PREPARE for r_2 in view v_2. Thus, no correct process could have sent a COMMIT message for r_2 in v_2, and thus it is impossible for a quorum to have sent COMMIT messages for for r_2 in v_2.

We have reached a contradiction in all four cases. This completes the proof of the theorem. □

C.2.3 Liveness

In this section, we provide informal arguments for the liveness of our protocol. We assume that the system is eventually synchronous and that correct processes propose values infinitely often. Intuitively, liveness is ensured by three mechanisms: (1) the view change mechanism, (2) checkpoints and (3) CTBcast summaries. First, we assume that CTBcast messages are not dropped and explain why liveness is ensured by the first two mechanisms. Then, we complete our explanation with the way CTBcast summaries help overcome the problem of dropped messages.

The view change mechanism ensures that at least one correct process is able to decide forever. Assume towards a contradiction that all correct processes stop deciding. Then, as long as they do not make progress, correct processes will change view thanks to the view change protocol in Algorithm C.2. Eventually, after the global stabilization time (GST), all correct processes are guaranteed to (1) reach a view v in which the leader is correct, and (2) communicate with each other in a timely manner. Thus, given that the timely collaboration of $f+1$ processes is enough for the common path described in Algorithm C.1 to be live, the correct replicas decide, hence a contradiction.

However, having a single correct process deciding infinitely often is not enough for the overall system to make progress as clients need to obtain a response from $f+1$ processes. The checkpoint mechanism guarantees that, if a correct process p decides on slots infinitely often, then all correct processes also make progress. This is because, in order to keep on making progress (and thus maintain correct processes under its control), the leader of a view is mandated to broadcast a `CHECKPOINT` message periodically. This message is then re-broadcast by the potentially single correct process in the view and, after GST, delivered by all correct processes. Because checkpoints are transferable, when another correct process receives a checkpoint, it is able to decide on the slots contained in the checkpoint and bring its application state up to speed with the latest decided requests.

Lastly, CTBcast summaries ensure that, if the system were to be reduced to only $f+1$ correct processes, they would be able to continue making progress in spite of CTBcast's delivery gaps. The worst scenario is arguably the one in which a correct process p used to make progress with f faulty processes before being let down by them, and the other f correct processes ending up with a gap in their CTBcast delivery of p's messages due to asynchrony. In this case, Algorithm C.3 ensures that p will not risk creating a gap before having obtained a summary to help overcoming it. Using this summary, p can let correct processes continue delivering its messages by convincing them that they will not violate the safety of the protocol due to missed messages. Moreover, p will always obtain a new summary: either it will be helped by Byzantine processes, or, after GST, it will get help from correct processes by combining the previous summary with the last t messages it broadcast.

Bibliography

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. 2005. Fault-scalable Byzantine fault-tolerant services. *SIGOPS Oper. Syst. Rev.* 39, 5 (Oct 2005), 59–74. https://doi.org/10.1145/1095809.1095817

[2] Advanced Micro Devices. 2022. AMD EPYC 9004 Series Server Processors. https://www.amd.com/en/processors/epyc-9004-series Accessed 2023-03-17.

[3] Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. 2018. Passing Messages While Sharing Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing* (Egham, United Kingdom) *(PODC '18)*. Association for Computing Machinery, New York, NY, USA, 51–60. https://doi.org/10.1145/3212734.3212741

[4] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. 2019. The Impact of RDMA on Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto, ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 409–418. https://doi.org/10.1145/3293611.3331601

[5] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. 2020. Microsecond Consensus for Microsecond Applications. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation* (Virtual Event) *(OSDI '20)*. USENIX Association, Berkeley, CA, USA, Article 34, 18 pages. https://www.usenix.org/conference/osdi20/presentation/aguilera

[6] Marcos K. Aguilera and Michael Walfish. 2009. No Time for Asynchrony. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems* (Monte Verità, Switzerland) *(HotOS '09)*. USENIX Association, Berkeley, CA, USA, 3. https://www.usenix.org/conference/hotos-xii/no-time-asynchrony

[7] Amitanand S. Aiyer, Lorenzo Alvisi, Rida A. Bazzi, and Allen Clement. 2008. Matrix Signatures: From MACs to Digital Signatures in Distributed Systems. In *Proceedings of the 22nd International Symposium on Distributed Computing* (Arcachon, France) *(DISC '08)*. Springer-Verlag, Berlin, Germany, 16–31. https://doi.org/10.1007/978-3-540-87779-0_2

[8] Noga Alon, Michael Merritt, Omer Reingold, Gadi Taubenfeld, and Rebecca N. Wright. 2005. Tight bounds for shared memory systems accessed by Byzantine processes. *Distributed Computing* 18, 2 (01 Nov 2005), 99–109. https://doi.org/10.1007/s00446-005-0125-8

[9] 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 Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the 13th European Conference on Computer Systems* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 30, 15 pages. https://doi.org/10.1145/3190508.3190538

[10] Behnaz Arzani, Selim Ciraci, Stefan Saroiu, Alec Wolman, Jack W. Stokes, Geoff Outhred, and Lechao Diwu. 2020. PrivateEye: Scalable and Privacy-Preserving Compromise Detection in the Cloud. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI '20)*. USENIX Association, Berkeley, CA, USA, 797–816. https://www.usenix.org/conference/nsdi20/presentation/arzani

[11] InfiniBand Trade Association. 2020. InfiniBand Architecture, General Specifications, Memory Placement Extensions. https://cw.infinibandta.org/document/dl/8594 Accessed 2023-03-17.

[12] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *J. ACM* 42, 1 (Jan. 1995), 124–142. https://doi.org/10.1145/200836.200869

[13] Hagit Attiya, Sweta Kumari, and Noa Schiller. 2021. Optimal Resilience in Systems That Mix Shared Memory and Message Passing. In *24th International Conference on Principles of Distributed Systems* (Virtual Event) *(OPODIS '20, Vol. 184)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:16. https://doi.org/10.4230/LIPIcs.OPODIS.2020.16

[14] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.* 32, 4, Article 12 (Jan 2015), 45 pages. https://doi.org/10.1145/2658994

[15] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. 2013. RBFT: Redundant Byzantine Fault Tolerance. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems* (Philadelphia, PA, USA) *(ICDCS '13)*. IEEE Computer Society, Los Alamitos, CA, USA, 297–306. https://doi.org/10.1109/ICDCS.2013.53

[16] Linux RDMA Authors. 2023. RDMA Benchmarking Utility. https://github.com/linux-rdma/perftest Accessed 2023-03-17.

[17] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia. 2021. Avocado: A Secure In-Memory Distributed Storage System. In *Proceedings of the 2021 USENIX Annual Technical Conference* (Virtual Event) *(USENIX ATC '21)*. USENIX Association, Berkeley, CA, USA, 65–79. https://www.usenix.org/conference/atc21/presentation/bailleu

[18] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures* (Pacific Grove, CA, USA) *(SPAA '16)*. Association for Computing Machinery, New York, NY, USA, 349–359. https://doi.org/10.1145/2935764.2935790

[19] Motti Beck and Michael Kagan. 2011. Performance Evaluation of the RDMA over Ethernet (RoCE) Standard in Enterprise Data Centers Infrastructure. In *Proceedings of the 3rd Workshop on Data Center - Converged and Virtual Ethernet Switching* (San Francisco, CA, USA) *(DC-CaVES '11)*. International Teletraffic Congress, San Francisco, CA, USA, 9–15. https://dl.acm.org/doi/10.5555/2043535.2043537

[20] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the 12th European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 222–237. https://doi.org/10.1145/3064176.3064213

[21] Naama Ben-David, Benjamin Y. Chan, and Elaine Shi. 2022. Revisiting the Power of Non-Equivocation in Distributed Protocols. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing* (Salerno, Italy) *(PODC '22)*. Association for Computing Machinery, New York, NY, USA, 450–459. https://doi.org/10.1145/3519270.3538427

[22] Naama Ben-David and Kartik Nayak. 2021. Brief Announcement: Classifying Trusted Hardware via Unidirectional Communication. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (Virtual Event) *(PODC '21)*. Association for Computing Machinery, New York, NY, USA, 191–194. https://doi.org/10.1145/3465084.3467948

[23] Michael Ben-Or. 1983. Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing* (Montreal, QC, Canada) *(PODC '83)*. Association for Computing Machinery, New York, NY, USA, 27–30. https://doi.org/10.1145/800221.806707

[24] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. 2015. SPHINCS: Practical Stateless Hash-Based Signatures. In *Advances in Cryptology – Proceedings of the 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques* (Sofia, Bulgaria) *(EUROCRYPT '15)*. Springer-Verlag, Berlin, Germany, 368–397. https://doi.org/10.1007/978-3-662-46800-5_15

[25] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Atlanta, GA, USA) *(DSN '14)*. IEEE Computer Society, NW Washington, DC, USA, 355–362. https://doi.org/10.1109/DSN.2014.43

[26] Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. 2009. Sharing Memory between Byzantine Processes Using Policy-Enforced Tuple Spaces. *IEEE Trans. Parallel Distrib. Syst.* 20, 3 (Mar 2009), 419–432. https://doi.org/10.1109/TPDS.2008.96

[27] Zohir Bouzid, Damien Imbs, and Michel Raynal. 2016. A Necessary Condition for Byzantine K-Set Agreement. *Inform. Process. Lett.* 116, 12 (Dec. 2016), 757–759. https://doi.org/10.1016/j.ipl.2016.06.009

[28] Gabriel Bracha. 1984. An Asynchronous [(n - 1)/3]-Resilient Consensus Protocol. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing* (Vancouver, BC, Canada) *(PODC '84)*. Association for Computing Machinery, New York, NY, USA, 154–162. https://doi.org/10.1145/800222.806743

[29] Gabriel Bracha. 1987. Asynchronous Byzantine Agreement Protocols. *Information and Computation* 75, 2 (Nov. 1987), 130–143. https://doi.org/10.1016/0890-5401(87)90054-X

[30] Gabriel Bracha and Sam Toueg. 1983. Resilient Consensus Protocols. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing* (Montreal, QC, Canada) *(PODC '83)*. Association for Computing Machinery, New York, NY, USA, 12–26. https://doi.org/10.1145/800221.806706

[31] Gabriel Bracha and Sam Toueg. 1985. Asynchronous Consensus and Broadcast Protocols. *J. ACM* 32, 4 (Oct. 1985), 824–840. https://doi.org/10.1145/4221.214134

[32] Eric Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2 (Feb. 2012), 23–29. https://doi.org/10.1109/MC.2012.37

[33] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. *Computing Research Repository (CoRR)* abs/1807.04938 (2018), 16 pages. https://doi.org/10.48550/ARXIV.1807.04938

[34] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (2nd ed.). Springer Publishing Company, New York, NY, USA. https://doi.org/10.1007/978-3-642-15260-3

[35] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and Efficient Asynchronous Broadcast Protocols. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '01)*. Springer-Verlag, Berlin, Germany, 524–541.

[36] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *Proceedings of the VLDB Endowment* 11, 11 (July 2018), 1604–1617. https://doi.org/10.14778/3236187.3236209

[37] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation* (New Orleans, LA, USA) *(OSDI '99)*. USENIX Association, Berkeley, CA, USA, 173–186. https://www.usenix.org/legacy/publications/library/proceedings/osdi99/castro.html

[38] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems* 20, 4 (Nov. 2002), 398–461. https://doi.org/10.1145/571637.571640

[39] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-Scale Acceleration Architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO '16)*. IEEE Computer Society, Los Alamitos, CA, USA, 7:1–7:13. https://doi.org/10.1109/MICRO.2016.7783710

[40] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-Scale Acceleration Architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, New York, NY, USA, Article 7, 13 pages.

[41] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. 1996. The Weakest Failure Detector for Solving Consensus. *J. ACM* 43, 4 (July 1996), 685–722. https://doi.org/10.1145/234533.234549

[42] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (March 1996), 225–267. https://doi.org/10.1145/226643.226647

[43] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. 2008. Diverse Replication for Single-Machine Byzantine-Fault Tolerance. In *Proceedings of the 2008 USENIX Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '08)*. USENIX Association, Berkeley, CA, USA, 287–292. https://www.usenix.org/conference/2008-usenix-annual-technical-conference/diverse-replication-single-machine-byzantine

[44] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested Append-Only Memory: Making Adversaries Stick to Their Word. *SIGOPS Operating Systems Review* 41, 6 (Oct. 2007), 189–204. https://doi.org/10.1145/1323293.1294280

[45] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. 2012. On the (Limited) Power of Non-Equivocation. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing* (Madeira, Portugal) *(PODC '12)*. Association for Computing Machinery, New York, NY, USA, 301–308. https://doi.org/10.1145/2332432.2332490

[46] Yann Collet. 2022. xxHash: Extremely fast non-cryptographic hash algorithm. https://github.com/Cyan4973/xxHash Accessed 2023-03-17.

[47] Compute Express Link Consortium. 2022. Compute Express Link (CXL) Specification, Revision 3.0. https://www.computeexpresslink.org/ Accessed 2023-03-17.

[48] Intel Corporation. 2016. Volume 3B: System Programming Guide, Part 2. In *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, San Jose, CA, USA. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf

[49] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2004. How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems* (Florianpolis, Brazil) *(SRDS '04)*. IEEE Computer Society, NW Washington, DC, USA, 174–183. https://doi.org/10.1109/RELDIS.2004.1353018

[50] Miguel Correia, Giuliana S. Veronese, and Lau Cheuk Lung. 2010. Asynchronous Byzantine Consensus with 2f+1 Processes. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (Sierre, Switzerland) *(SAC '10)*. Association for Computing Machinery, New York, NY, USA, 475–480. https://doi.org/10.1145/1774088.1774187

[51] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. https://eprint.iacr.org/2016/086 Accessed 2023-03-17.

[52] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. 2021. SmashEx: Smashing SGX Enclaves Using Exceptions. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 779–793. https://doi.org/10.1145/3460120.3484821

[53] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus. In *Proceedings of the 17th European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 34–50. https://doi.org/10.1145/3492321.3519594

[54] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[55] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[56] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. 2004. The Weakest Failure Detectors to Solve Certain Fundamental Problems in Distributed Computing. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Distributed Computing* (St. John's, NL, Canada) *(PODC '04)*. Association for Computing Machinery, New York, NY, USA, 338–346. https://doi.org/10.1145/1011767.1011818

[57] Linux Kernel Developers. 2022. NO_HZ: Reducing Scheduling-Clock Ticks. https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt Accessed 2023-03-17.

[58] Whitfield Diffie and Martin E. Hellman. 1976. New directions in cryptography. *IEEE Transactions on Information Theory* 22, 6 (Nov. 1976), 644–654. https://doi.org/10.1109/TIT.1976.1055638

[59] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. 2022. Detecting silent data corruptions in the wild. *Computing Research Repository (CoRR)* abs/2203.08989 (2022), 7 pages. https://doi.org/10.48550/arXiv.2203.08989

[60] Dan Dobre and Neeraj Suri. 2006. One-Step Consensus with Zero-Degradation. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks* (Philadelphia, PA, USA) *(DSN '06)*. IEEE Computer Society, NW Washington, DC, USA, 137–146. https://doi.org/10.1109/DSN.2006.55

[61] Danny Dolev. 1982. The Byzantine Generals Strike Again. *Journal of Algorithms* 3, 1 (1982), 14–30.

[62] Danny Dolev and Rüdiger Reischuk. 1985. Bounds on Information Exchange for Byzantine Agreement. *J. ACM* 32, 1 (Jan. 1985), 191–204. https://doi.org/10.1145/2455.214112

[63] Travis Downs. 2022. A Benchmark for Low-level CPU Micro-architectural Features. https://github.com/travisdowns/uarch-bench Accessed 2023-03-17.

[64] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation* (Seattle, WA, USA) *(NSDI '14)*. USENIX Association, Berkeley, CA, USA, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87

[65] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles* (Monterey, CA, USA) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[66] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (April 1988), 288–323. https://doi.org/10.1145/42282.42283

[67] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. 2016. Bitcoin-NG: A Scalable Blockchain Protocol. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI '16)*. USENIX Association, Berkeley, CA, USA, 45–59. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eyal

[68] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. 2021. Security Vulnerabilities of SGX and Countermeasures: A Survey. *Comput. Surveys* 54, 6, Article 126 (July 2021), 36 pages. https://doi.org/10.1145/3456631

[69] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI '18)*. USENIX Association, Berkeley, CA, USA, 51–64. https://www.usenix.org/conference/nsdi18/presentation/firestone

[70] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (apr 1985), 374–382. https://doi.org/10.1145/3149.214121

[71] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 51–68. https://doi.org/10.1145/3132747.3132757

[72] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News* 33, 2 (June 2002), 51–59. https://doi.org/10.1145/564585.564601

[73] Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (Portland, OR, USA) *(DSN '19)*. IEEE Computer Society, NW Washington, DC, USA, 568–580. https://doi.org/10.1109/DSN.2019.00063

[74] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *Proceedings of the 2022 USENIX Annual Technical Conference* (Carlsbad, CA, USA) *(USENIX ATC '22)*. USENIX Association, Berkeley, CA, USA, 287–294. https://www.usenix.org/conference/atc22/presentation/gouk

[75] Tomasz Gromadzki and Jan Marian Michalski. 2019. Persistent Memory Replication Over Traditional RDMA Part 1: Understanding Remote Persistent Memory. https://www.intel.com/content/www/us/en/developer/articles/technical/persistent-memory-replication-over-traditional-rdma-part-1-understanding-remote-persistent.html Accessed 2023-03-17.

[76] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. 2018. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. *ACM Transactions on Storage* 14, 3, Article 23 (Oct. 2018), 26 pages. https://doi.org/10.1145/3242086

[77] Divya Gupta, Lucas Perronne, and Sara Bouchenak. 2016. BFT-Bench: A Framework to Evaluate BFT Protocols. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering* (Delft, The Netherlands) *(ICPE '16)*. Association for Computing Machinery, New York, NY, USA, 109–112. https://doi.org/10.1145/2851553.2858667

[78] Vassos Hadzilacos. 1985. *Issues of Fault Tolerance in Concurrent Computations.* Ph. D. Dissertation. Harvard University, Cambridge, MA, USA. AAI 8520209.

[79] Vassos Hadzilacos, Xing Hu, and Sam Toueg. 2020. Optimal Register Construction in M&M Systems. In *23rd International Conference on Principles of Distributed Systems* (Neuchâtel, Switzerland) *(OPODIS '19, Vol. 153)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 28:1–28:16. https://doi.org/10.4230/LIPIcs.OPODIS.2019.28

[80] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (Jan. 1991), 124–149. https://doi.org/10.1145/114005.102808

[81] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA) *(NSDI '11)*. USENIX Association, Berkeley, CA, USA, 295–308. https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center

[82] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. 2021. Cores That Don't Count. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems* (Ann Arbor, MI, USA) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/3458336.3465297

[83] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 150–155. https://doi.org/10.1145/3102980.3103005

[84] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '10)*. USENIX Association, Berkeley, CA, USA, 11 pages. https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems

[85] Danga Interactive. 2022. Memcached. https://memcached.org/ Accessed 2023-03-17.

[86] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent Distributed Storage. *Proceedings of the VLDB Endowment* 10, 11 (Aug. 2017), 1202–1213. https://doi.org/10.14778/3137628.3137632

[87] Hai Jin, Rajkumar Buyya, and Toni Cortes. 2002. An Introduction to the InfiniBand Architecture. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications* (1st ed.). John Wiley and Sons, Inc., Hoboken, NJ, USA, 616–632. https://doi.org/10.1109/9780470544839

[88] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation* (Renton, WA, USA) *(NSDI '18)*. USENIX Association, Berkeley, CA, USA, 35–49. https://www.usenix.org/conference/nsdi18/presentation/jin

[89] Simon Josefsson and Ilari Liusvaara. 2017. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032. https://doi.org/10.17487/RFC8032

[90] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI '19)*. USENIX Association, Berkeley, CA, USA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[91] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, IL, USA) *(SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 295–306. https://doi.org/10.1145/2619239.2626299

[92] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[93] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-Efficient Byzantine Fault Tolerance. In *Proceedings of the 7th European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/2168836.2168866

[94] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 201–217. https://doi.org/10.1145/3373376.3378496

[95] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography* (2nd ed.). Chapman and Hall/CRC Press, Boca Raton, FL, USA.

[96] Idit Keidar and Sergio Rajsbaum. 2003. On the Cost of Fault-Tolerant Consensus When There Are No Faults – A Tutorial. In *Proceedings of the 1st Latin-American Symposium on Dependable Computing* (Sao Paulo, Brazil) *(LADC 2003)*. Springer-Verlag, Berlin, Germany, 366–368. https://doi.org/10.1007/978-3-540-45214-0_29

[97] Colin King. 2023. stress-ng: A Tool to Load and Stress a Computer System. https://github.com/ColinIanKing/stress-ng Accessed 2023-03-17.

[98] Patrick Knebel, Dan Berkram, Al Davis, Darel Emmot, Paolo Faraboschi, and Gary Gostin. 2019. Gen-Z Chipset for Exascale Fabrics. In *2019 IEEE Hot Chips 31 Symposium (HCS)* (Cupertino, CA, USA). IEEE Computer Society, NW Washington, DC, USA, 1–22. https://doi.org/10.1109/HOTCHIPS.2019.8875646

[99] Marios Kogias and Edouard Bugnion. 2020. HovercRaft: Achieving Scalability and Fault-tolerance for microsecond-scale Datacenter Services. In *Proceedings of the 15th European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 25, 17 pages. https://doi.org/10.1145/3342195.3387545

[100] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC '16)*. USENIX Association, Berkeley, CA, USA, 279–296. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kogias

[101] Martijn Koot and Fons Wijnhoven. 2021. Usage impact on data center electricity needs: A system dynamic forecasting model. *Applied Energy* 291 (June 2021), 116798. https://doi.org/10.1016/j.apenergy.2021.116798

[102] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems* 27, 4, Article 7 (Jan. 2010), 39 pages. https://doi.org/10.1145/1658357.1658358

[103] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A Distributed Messaging System for Log Processing. In *6th International Workshop on Networking Meets Databases* (Athens, Greece) *(NetDB'11)*. Association for Computing Machinery, New York, NY, USA, 7 pages. https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf

[104] Ajay D. Kshemkalyani and Mukesh Singhal. 2008. *Distributed Computing: Principles, Algorithms, and Systems* (1 ed.). Cambridge University Press, New York, NY, USA.

[105] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922

[106] Leslie Lamport. 1977. Concurrent Reading and Writing. *Commun. ACM* 20, 11 (Nov. 1977), 806–811. https://doi.org/10.1145/359863.359878

[107] Leslie Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering* 3, 2 (March 1977), 125–143. https://doi.org/10.1109/TSE.1977.229904

[108] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[109] Leslie Lamport. 1983. The Weak Byzantine Generals Problem. *J. ACM* 30, 3 (July 1983), 668–676. https://doi.org/10.1145/2402.322398

[110] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[111] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19, 2 (Oct. 2006), 79–103. https://doi.org/10.1007/s00446-006-0005-x

[112] Leslie Lamport and Michael Fischer. 1982. *Byzantine Generals and Transaction Commit Protocols.* Technical Report TRO-62. SRI International, Menlo Park, CA, USA. https://www.microsoft.com/en-us/research/publication/byzantine-generals-transaction-commit-protocols/

[113] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* 4, 3 (July 1982), 382–401. https://doi.org/10.1145/357172.357176

[114] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles* (Virtual Event) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 488–504. https://doi.org/10.1145/3477132.3483561

[115] Youngmoon Lee, Hassan Al Maruf, Mosharaf Chowdhury, and Kang G. Shin. 2019. Mitigating the Performance-Efficiency Tradeoff in Resilient Memory Disaggregation. *Computing Research Repository (CoRR)* abs/1910.09727 (2019), 14 pages. https://doi.org/10.48550/ARXIV.1910.09727

[116] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, GA, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 517–530. https://doi.org/10.1145/2872362.2872374

[117] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. 2013. Improving Availability in Distributed Systems with Failure Informers. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation* (Lombard, IL, USA) *(NSDI '13)*. USENIX Association, Berkeley, CA, USA, 427–442. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/leners

[118] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. 2015. Taming Uncertainty in Distributed Systems with Help from the Network. In *Proceedings of the 10th European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 9, 16 pages. https://doi.org/10.1145/2741948.2741976

[119] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. 2011. Detecting Failures in Distributed Systems with the FALCON Spy Network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 279–294. https://doi.org/10.1145/2043556.2043583

[120] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI '09)*. USENIX Association, Berkeley, CA, USA, 1–14. https://www.usenix.org/conference/nsdi-09/trinc-small-trusted-hardware-large-distributed-systems

[121] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High Precision Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 44–58. https://doi.org/10.1145/3341302.3342085

[122] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. 2019. Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing. *IEEE Trans. Comput.* 68, 1 (Jan. 2019), 139–151. https://doi.org/10.1109/TC.2018.2860009

[123] Isis Agora Lovecruft and Henry De Valence. 2022. ed25519-dalek: Fast and efficient Rust implementation of ed25519 key generation, signing, and verification in Rust. https://github.com/dalek-cryptography/ed25519-dalek Accessed 2023-03-17.

[124] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, CA, USA. https://doi.org/10.5555/2821576

[125] Mads Frederik Madsen and Søren Debois. 2020. On the Subject of Non-Equivocation: Defining Non-Equivocation in Synchronous Agreement Systems. In *Proceedings of the 39th ACM Symposium on Principles of Distributed Computing* (Virtual Event) *(PODC '20)*. Association for Computing Machinery, New York, NY, USA, 159–168. https://doi.org/10.1145/3382734.3405731

[126] Dahlia Malkhi, Michael Merritt, Michael K. Reiter, and Gadi Taubenfeld. 2003. Objects Shared by Byzantine Processes. *Distributed Computing* 16, 1 (Feb. 2003), 37–48. https://doi.org/10.1007/s00446-002-0075-3

[127] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (San Diego, CA, USA) *(OSDI '08)*. USENIX Association, Berkeley, CA, USA, 369–384. https://www.usenix.org/conference/osdi-08/mencius-building-efficient-replicated-state-machines-wans

[128] Jean-Philippe Martin and Lorenzo Alvisi. 2006. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (July 2006), 202–215. https://doi.org/10.1109/TDSC.2006.35

[129] Ines Messadi, Markus Horst Becker, Kai Bleeke, Leander Jehl, Sonia Ben Mokhtar, and Rüdiger Kapitza. 2022. SplitBFT: Improving Byzantine Fault Tolerance Safety Using Trusted Compartments. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference* (Quebec, QC, Canada) *(Middleware '22)*. Association for Computing Machinery, New York, NY, USA, 56–68. https://doi.org/10.1145/3528535.3531516

[130] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. 2015. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Portland, OR, USA) *(SIGMETRICS '15)*. Association for Computing Machinery, New York, NY, USA, 177–190. https://doi.org/10.1145/2745844.2745848

[131] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. 2018. A Large Scale Study of Data Center Network Reliability. In *Proceedings of the Internet Measurement Conference 2018* (Boston, MA, USA) *(IMC '18)*. Association for Computing Machinery, New York, NY, USA, 393–407. https://doi.org/10.1145/3278532.3278566

[132] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/2976749.2978399

[133] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference* (San Jose, CA, USA) *(USENIX ATC '13)*. USENIX Association, Berkeley, CA, USA, 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[134] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (Farminton, PA, USA) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 358–372. https://doi.org/10.1145/2517349.2517350

[135] Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. 2014. Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing* (Paris, France) *(PODC '14)*. Association for Computing Machinery, New York, NY, USA, 2–9. https://doi.org/10.1145/2611462.2611468

[136] Sape Mullender (Ed.). 1993. *Distributed Systems* (2nd ed.). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

[137] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf Accessed 2023-03-17.

[138] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. 2016. SSD Failures in Datacenters: What, When and Why?. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science* (Antibes Juan-les-Pins, France) *(SIGMETRICS '16)*. Association for Computing Machinery, New York, NY, USA, 407–408. https://doi.org/10.1145/2896377.2901489

[139] Eric Newhuis. 2022. Liquibook: Open source order matching engine. https://github.com/enewhuis/liquibook Accessed 2023-03-17.

[140] NVIDIA. 2022. NVIDIA ConnectX-6 DX Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectX-6-dx-datasheet.pdf Accessed 2023-03-17.

[141] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. 2022. BLAKE3. https://github.com/BLAKE3-team/BLAKE3 Accessed 2023-03-17.

[142] David Oppenheimer, Archana Ganapathi, and David A. Patterson. 2003. Why Do Internet Services Fail, and What Can Be Done about It?. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4* (Seattle, WA, USA) *(USITS '03)*. USENIX Association, Berkeley, CA, USA, 15 pages. https://www.usenix.org/conference/usits-03/why-do-internet-services-fail-and-what-can-be-done-about-it

[143] Marshall Pease, Robert Shostak, and Leslie Lamport. 1980. Reaching Agreement in the Presence of Faults. *J. ACM* 27, 2 (April 1980), 228–234. https://doi.org/10.1145/322186.322188

[144] René Peinl, Florian Holzschuher, and Florian Pfitzer. 2016. Docker Cluster Management for the Cloud-Survey Results and Own Solution. *Journal of Grid Computing* 14, 2 (01 June 2016), 265–282. https://doi.org/10.1007/s10723-016-9366-y

[145] Kenneth J. Perry and Sam Toueg. 1986. Distributed Agreement in the Presence of Processor and Communication Faults. *IEEE Transactions on Software Engineering* 12, 3 (March 1986), 477–482. https://doi.org/10.1109/TSE.1986.6312888

[146] Brandon Philips, Alex Polvi, and Xiang Li. 2022. Etcd. https://etcd.io Accessed 2023-03-17.

[147] Marius Poke and Torsten Hoefler. 2015. DARE: High-performance state machine replication on RDMA networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing* (Portland, Oregon, USA) *(HPDC '15)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/2749246.2749267

[148] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780

[149] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *ACM SIGARCH Computer Architecture News* 42, 3 (June 2014), 13–24. https://doi.org/10.1145/2678373.2665678

[150] Michel Raynal and Jiannong Cao. 2019. One for All and All for One: Scalable Consensus in a Hybrid Communication Model. In *Proceedings of the 2019 IEEE 39th International Conference on Distributed Computing Systems* (Dallas, TX, USA) *(ICDCS '19)*. IEEE Computer Society, Los Alamitos, CA, USA, 464–471. https://doi.org/10.1109/ICDCS.2019.00053

[151] Red Hat. 2020. RHEL for Real Time Timestamping. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/chap-timestamping Accessed 2023-03-17.

[152] Michael K. Reiter. 1994. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security* (Fairfax, Virginia, USA) *(CCS '94)*. Association for Computing Machinery, New York, NY, USA, 68–80. https://doi.org/10.1145/191177.191194

[153] Benjamin Rothenberger, Konstantin Taranov, Adrian Perrig, and Torsten Hoefler. 2021. ReDMArk: Bypassing RDMA Security Mechanisms. In *30th USENIX Security Symposium (USENIX Security '21)* (Virtual Event). USENIX Association, Berkeley, CA, USA, 4277–4292. https://www.usenix.org/conference/usenixsecurity21/presentation/rothenberger

[154] Signe Rüsch, Ines Messadi, and Rüdiger Kapitza. 2018. Towards Low-Latency Byzantine Agreement Protocols Using RDMA. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)* (Luxembourg, Luxembourg). IEEE Computer Society, NW Washington, DC, USA, 146–151. https://doi.org/10.1109/DSN-W.2018.00054

[155] J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems* 2, 4 (Nov 1984), 277–288. https://doi.org/10.1145/357401.357402

[156] Salvatore Sanfilippo. 2022. Redis. https://github.com/redis/redis Accessed 2023-03-17.

[157] Fred B. Schneider. 1984. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems* 2, 2 (May 1984), 145–154. https://doi.org/10.1145/190.357399

[158] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *Comput. Surveys* 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167

[159] Omid Shahmirzadi, Sergio Mena, and Andre Schiper. 2009. Relaxed Atomic Broadcast: State-Machine Replication Using Bounded Memory. In *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems* (Niagara Falls, NY, USA) *(SRDS '09)*. IEEE Computer Society, NW Washington, DC, USA, 3–11. https://doi.org/10.1109/SRDS.2009.25

[160] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. 2011. Sharing the Data Center Network. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI '11)*. USENIX Association, Berkeley, CA, USA, 309–322. https://www.usenix.org/conference/nsdi11/sharing-data-center-network

[161] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, NW Washington, DC, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[162] Swaminathan Sivasubramanian. 2012. Amazon DynamoDB: A Seamlessly Scalable Non-Relational Database Service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 729–730. https://doi.org/10.1145/2213836.2213945

[163] T. K. Srikanth and Sam Toueg. 1987. Simulating Authenticated Broadcasts to Derive Simple Fault-Tolerant Algorithms. *Distributed Computing* 2, 2 (June 1987), 80–94. https://doi.org/10.1007/BF01667080

[164] Anish Sukumaran and Vincent G. Nicotra. 2018. Lease-based Leader Election System. US Patent 9984140.

[165] Konstantin Taranov, Benjamin Rothenberger, Adrian Perrig, and Torsten Hoefler. 2020. sRDMA: Efficient NIC-Based Authentication and Encryption for Remote Direct Memory Access. In *Proceedings of the 2020 USENIX Annual Technical Conference* (Virtual Event) *(USENIX ATC '20)*. USENIX Association, Berkeley, CA, USA, Article 47, 14 pages. https://www.usenix.org/conference/atc20/presentation/taranov

[166] Mellanox Technologies. 2015. RDMA Aware Networks Programming User Manual. Rev 1.7. https://docs.nvidia.com/networking/spaces/viewspace.action?key=RDMAAwareProgrammingv17 Accessed 2023-03-17.

[167] Mellanox Technologies. 2022. VMA: Linux user space library for network socket acceleration based on RDMA compatible network adaptors. https://github.com/Mellanox/libvma Accessed 2023-03-17.

[168] Shahar Timnat and Erez Petrank. 2014. A Practical Wait-Free Simulation for Lock-Free Data Structures. *SIGPLAN Notices* 49, 8 (Feb. 2014), 357–368. https://doi.org/10.1145/2692916.2555261

[169] Sam Toueg. 1984. Randomized Byzantine Agreements. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing* (Vancouver, BC, Canada) *(PODC '84)*. Association for Computing Machinery, New York, NY, USA, 163–178. https://doi.org/10.1145/800222.806744

[170] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of the 2020 USENIX Annual Technical Conference* (Virtual Event) *(USENIX ATC '20)*. USENIX Association, Berkeley, CA, USA, Article 3, 16 pages. https://www.usenix.org/conference/atc20/presentation/tsai

[171] Maarten van Steen and Andrew S. Tanenbaum. 2016. A brief introduction to distributed systems. *Computing* 98, 10 (01 Oct. 2016), 967–1009. https://doi.org/10.1007/s00607-016-0508-7

[172] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Comput.* 62, 1 (Jan. 2013), 16–30. https://doi.org/10.1109/TC.2011.221

[173] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. 2017. APUS: Fast and scalable Paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 94–107. https://doi.org/10.1145/3127479.3128609

[174] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2022. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. *Proceedings of the VLDB Endowment* 16, 1 (Sept. 2022), 15–22. https://doi.org/10.14778/3561261.3561263

[175] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-Enabled Distributed Transactions: Hybrid is Better. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI '18)*. USENIX Association, Berkeley, CA, USA, 233–251. https://www.usenix.org/conference/osdi18/presentation/wei

[176] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles* (Monterey, CA, USA) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 87–104. https://doi.org/10.1145/2815400.2815419

[177] Tian Yang, Robert Gifford, Andreas Haeberlen, and Linh Thi Xuan Phan. 2019. The Synchronous Data Center. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 142–148. https://doi.org/10.1145/3317550.3321442

[178] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. 2003. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 253–267. https://doi.org/10.1145/945445.945470

[179] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto, ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591

[180] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proceedings of the VLDB Endowment* 10, 6 (Feb. 2017), 685–696. https://doi.org/10.14778/3055330.3055335

[181] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI '22)*. USENIX Association, Berkeley, CA, USA, 55–71. https://www.usenix.org/conference/osdi22/presentation/zhou-yang

[182] Danyang Zhuo, Qiao Zhang, Dan R. K. Ports, Arvind Krishnamurthy, and Thomas Anderson. 2014. Machine Fault Tolerance for Reliable Datacenter Systems. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (Beijing, China) *(APSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 3, 7 pages. https://doi.org/10.1145/2637166.2637235

Athanasios Xygkis

Distributed Computing Laboratory
School of Computer and Communication Sciences
EPFL
athanasios.xygkis@epfl.ch
xygkis

I am interested in distributed computing, in particular in the intersection between theory and practice. My focus revolves around consensus, availability, fault tolerance, and failure detection of low-latency data center distributed systems.

Education

2018–2023 **PhD in Computer Science**,
École Polytechnique Fédérale de Lausanne (EPFL)
- Thesis: Reliable Microsecond-Scale Distributed Computing
- Advisor: Prof. Rachid Guerraoui

2011–2017 **BSc & MSc in Electrical and Computer Engineering**,
National Technical University of Athens (NTUA)
- Major: Computer Science, Control Theory
- Grade: 9.30/10
- Thesis: Implementation of Convolutional Neural Networks in Embedded Architectures
- Supervisor: Prof. Dimitrios Soudris

Internships

2017–2018 **Intel Corporation**, *Perceptual Computing Group, Movidius R&D, Ireland*
- Topic: Compilation of Convolutional Neural Networks for Intel's Myriad SoC
- Supervisor: Dr. Sofiane Yous

Research Output

Peer-reviewed Conference Papers (author names in alphabetical order)

2023 **uBFT: Microsecond-Scale BFT using Disaggregated Memory**, *ASPLOS '23*
Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi

2022 **uKharon: A Membership Service for Microsecond Applications**, *USENIX ATC '22*
Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo.

2021 **Frugal Byzantine Computing**, *DISC '21*
Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Dalia Papuc, Athanasios Xygkis, and Igor Zablotchi

2020 **Microsecond Consensus for Microsecond Applications**, *OSDI '20*
Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi

2020 **Online Payments by Merely Broadcasting Messages**, *DSN '20*
Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis

Conference Papers under Submission (author names in alphabetical order)

2023 **Pony: Breaking the Barrier of Signatures in Data Centers**
Marcos K. Aguilera, Clément Burgelin, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi

Conference Presentations

2023 uBFT: Microsecond-scale BFT using disaggregated memory, *ASPLOS '23*

2020 Online payments by merely broadcasting messages, *DSN '20*

Conference Posters

2023 uBFT: Microsecond-scale BFT using disaggregated memory, *ASPLOS '23*

Languages

Greek	Native Proficiency	French	Limited Working Proficiency
English	Full Professional Proficiency	Spanish	Elementary Proficiency

Honors & Awards

2022 *Patent*: For the work at USENIX ATC '22

2020 *Award*: Runner up for Best Paper Award at DSN '20

2019 *Fellowship*: Joining EPFL's PhD program

2016 *Scholarship*: Excellence in mathematics during my undergraduate studies at NTUA

Teaching

Teaching Assistant

2019–2022 Distributed Algorithms, *Graduate Course, EPFL*

2021–2022 Systems for Data Science, *Graduate Course, EPFL*

2019 Object Oriented Programming, *Undergraduate Course, EPFL*

Mentoring

2023 Nitish Ravishankar
Safe Rust Abstractions for RDMA, MSc Semester Project, EPFL

2022 Lovro Nuic
Efficient State Transfer using CRIU and RDMA, MSc Semester Project, EPFL

2021 Lovro Nuic
Offloading Turing-Complete Functions on RDMA NICs, MSc Semester Project, EPFL

2020–2021 Kristian Brünn
Byzantine Fault Tolerant State Machine Replication with RDMA, MSc Thesis, EPFL