UNIVERSITÄT PASSAU

Fakultät für Informatik und Mathematik

Towards Fast and Adaptive Byzantine State Machine Replication for Planetary-Scale Systems

Christian Berger

Dissertation eingereicht an der Fakultät für Informatik und Mathematik der Universität Passau zur Erlangung des Grades eines Doktors der Naturwissenschaften

Betreuer:	Prof. Dr. Hans P. Reiser
Zweitgutachter:	Prof. Dr. Jérémie Decouchant

Rigorosum am:	28.10.2024

Passau, May 2024

Abstract

State machine replication (SMR) is a classical approach for building resilient distributed systems. In Byzantine fault-tolerant (BFT) systems, no concrete assumptions are made about the behavior of faulty replicas. With the advancement of distributed ledger technologies (DLT), planetary-scale BFT SMR ist becoming practical and necessary as it can serve as a consensus primitive to keep the ledger consistent. In our view, the alignment of BFT SMR to DLT brings new challenges, for instance the scalability aspect, where recent research works less frequently address latency improvements than throughput improvements. Further challenges include the geographic dispersion of replicas within a planetary-scale system and the need of a BFT SMR protocol to react to environmental changes during runtime.

This thesis has the objective to improve BFT SMR for planetary-scale systems by lowering the protocol latency observed by clients and by making the BFT SMR system adaptive, i.e., enabling replicas to react to perceived changes such as changing network characteristics or faulty replicas.

As a first contribution of this thesis, we discover that fast, consensus-free (read-only) operations is a flawed optimization in seminal BFT SMR frameworks, such as PBFT and BFT-SMaRt. We explain how the read-only optimization can violate the protocol's liveness by showing an attack and then present a solution that makes the overall, optimized protocol both live and linearizable.

The second contribution is Adaptive Wide-Area Replication (AWARE), which enables a geo-replicated system to adapt to its environment, thus improving the geographical scalability of consensus if replicas are dispersed across the world. Essentially, AWARE is an automated and dynamic voting-weight tuning and leader positioning scheme, which supports the emergence of fast consensus quorums in the system and builds upon previous work, the WHEAT protocol. AWARE combines reliable self-monitoring with a consensus latency prediction model, thus striving to minimize the system's consensus latency at runtime, which subsequently results in latency improvements observed by clients scattered across the globe, which we validate through experiments.

The third contribution presents FlashConsensus, a protocol derived from AWARE, that also adjusts the resilience threshold. The core idea is the tentative use of a lower resilience threshold which leads to smaller consensus quorums and thus consensus acceleration in common-case scenarios where we expect only few faulty replicas. FlashConsensus achieves threat-level awareness through the incorporation of two modes of operation and BFT forensic support and guarantees liveness and linearizability under optimal resilience. Moreover, FlashConsensus allows for client-side speculation by using incremental consistency guarantees to further lower request latency.

Additionally, we investigate on the question whether we can reason about the performance of large-scale systems utilizing simulations. We discover, that we can faithfully forecast the performance of BFT protocols by plugging real protocol implementations into a high-performance network simulator. For instance, simulation results reveal that, using 51 replicas scattered across the planet, FlashConsensus can finalize operations in less than 0.4 s, which is half of the time required for a PBFT-like protocol in the same network, and matching the latency of this protocol running on the best possible internet links (transmitting at 67% of the speed of light).

Keywords: Byzantine fault tolerance, state machine replication, consensus, adaptiveness, weighted replication, low latency, planetary-scale, geographic scalability

Zusammenfassung

Die Zustandsmaschinenreplikation (ZMR) ist ein klassischer Ansatz für zuverlässige verteilte Systeme. In Byzantinischen fehlertoleranten (BFT) Systemen werden keine konkreten Annahmen über das Verhalten von fehlerhaften Replikaten gemacht. Mit dem Voranschreiten der sog., „Distributed Ledger Technologien“ (DLT) wird planetare BFT ZMR praktikabel und notwendig, da sie als Konsensusprimitiv dienen kann, um die Konsistenz einer Blockchain aufrechtzuerhalten.

Unserer Ansicht nach bringt die Ausrichtung von BFT ZMR an DLT neue Herausforderungen mit sich, z.B. den Skalierbarkeitsaspekt, bei dem jüngste Forschungsarbeiten seltener Latenzverbesserungen als Durchsatzverbesserungen untersuchen. Weitere Herausforderungen umfassen die geografische Verteilung von Replikaten innerhalb eines planetaren Systems sowie die Notwendigkeit eines BFT ZMR Protokolls während der Laufzeit auf Veränderungen zu reagieren.

Diese Dissertation verfolgt das Ziel, die BFT ZMR für planetare Systeme durch Senkung der von Clients beobachteten Protokolllatenz zu verbessern und die BFT ZMR-Systeme anpassungsfähig zu machen, indem Replikate auf wahrgenommene Änderungen wie sich ändernde Netzwerkcharakteristiken oder fehlerhafte Replikate reagieren können.

Als erster Beitrag dieser Dissertation zeigen wir, dass schnelle, konsensusfreie (nur-lesende) Operationen in grundlegenden BFT ZMR Protokollen, wie PBFT und BFT-SMaRt, eine fehlerhafte Optimierung sind. Wir erklären, wie die nur-lesende Optimierung die Liveness (Verfügbarkeit von Operationen) des Protokolls verletzen kann, indem wir einen Angriff präsentieren und dann eine Lösung vorschlagen, die das insgesamt optimierte Protokoll sowohl live (verfügbar) als auch linearisierbar („linearizable“– streng konsistent) macht.

Der zweite Beitrag ist Adaptive Wide-Area Replication (AWARE), mit der ein geo-repliziertes System sich an seine Umgebung anpassen kann und damit die geografische Skalierbarkeit des Konsensus verbessert, wenn Replikate auf der ganzen Welt verteilt sind. Im Wesentlichen ist AWARE ein automatisches und dynamisches Stimmgewichtseinstellungs- und Anführerpositionierungs-schema, das die Entstehung schneller Konsensusquoren im System unterstützt und auf früheren Arbeiten, wie dem WHEAT-Protokoll, aufbaut. AWARE kombiniert zuverlässige Selbstüberwachung mit einem Konsensuslatenzvorhersagemodell und strebt danach, die Konsensuslatenz des Systems zur Laufzeit zu minimieren, was letztendlich zu Latenzgewinnen führt, die von Clients auf der ganzen Welt beobachtbar sind, was wir durch Experimente bestätigen.

Der dritte Beitrag stellt FlashConsensus vor, ein Protokoll, das aus AWARE abgeleitet ist und auch die Resilienzschwelle anpasst. Die Kernidee ist die vorübergehende Verwendung einer niedrigeren Resilienzschwelle, die zu kleineren Konsensusquoren und damit zu einer Beschleunigung des Konsensus in häufigen Fällen führt, in denen nur wenige fehlerhafte Replikate erwartet werden. FlashConsensus erkennt und reagiert auf Bedrohungen durch die Einbeziehung von zwei Betriebsarten und BFT-Forensikunterstützung und garantiert Liveness sowie Konsistenz unter optimaler Resilienz. Darüber hinaus ermöglicht es die Spekulation auf Clientseite durch die Verwendung inkrementeller Konsistenzgarantien, um die Clientlatenz weiter zu senken.

Zusätzlich werden wir uns mit der Frage beschäftigen, ob wir die Performanz von groß-skalierten Systemen mittels Simulationen beurteilen können. Wir stellen fest, dass wir die Performanz von BFT Protokollen durch das Einstöpseln von echten Protokollimplementierungen in einen hochleistungsfähigen Netzwerksimulator zuverlässig vorhersagen können. Beispielsweise zeigen Simulationen, dass FlashConsensus mit 51 Replikaten, die auf der ganzen Welt verteilt sind, Operationen in weniger als 0,4 s linearisierbar verarbeiten kann, was die Hälfte der Zeit ist, die für ein PBFT-ähnliches Protokoll im gleichen Netzwerk erforderlich ist und ähnlich schnell ist wie dieses Protokoll mit bestmöglichen Internetverbindungen (Übertragung mit 67% der Lichtgeschwindigkeit).

Stichwörter: Byzantinische Fehlertoleranz, Zustandsmaschinenreplikation, Konsensus, Anpassungsfähigkeit, Gewichtete Replikation, niedrige Latenz, planetarer Maßstab, geografische Skalierbarkeit

Acknowledgements

First of all I want to express my deep thankfulness to my supervisor and mentor, Professor Hans P. Reiser. Not only did I complete my Master thesis under his guidance, but he also motivated me to start working on a PhD project and provided me with the opportunity to work at the university. During my PhD journey, he steadily guided me with his vast expertise and reliable support. It is thanks to his mentorship, that I could shape and grow into a researcher, learning the essential skills needed in academia.

I also owe a big thanks to my second examiner, Professor Jérémie Decouchant, for his prompt willingness to support my thesis project as the second examiner and provide insightful feedback.

Special thanks to Professors Rüdiger Kapitza and Leander Jehl for their continual guidance and support during the last years we shared on our research project. With their experience, they helped me to improve my research because I could steadily rely on their feedback. I am also deeply grateful to Professor Alysson Bessani for collaborating with me on many papers, his expertise and his mentorship have really broadened my understanding of the field. Moreover, I would also like to thank Professor Franz J. Hauck for proof reading and offering insightful feedback on my thesis.

Special acknowledgment is due to Siglinde Böck for her perpetual assistance and support during all the years I worked at the university. I extend my sincere gratitude to my friends and colleagues, who helped me and made the time I spent at my workplace always so cheerful: Philipp Eichhammer, Stewart Sentanoe, Thomas Dangl, Felix Klement, Simon Unger, Florian Frank, Dominik Püllen, Kailun Sha, Benjamin Taubmann, Henrich Pöhls, Cornelius Brand, Thorben Voss and Andreas Scheibenpflug.

It really motivated me and gave me strength to count on their friendship, support and encouragement throughout this journey.

I also want to thank my girlfriend, Sabine Bals, for her dedicated support and care. Her love and encouragement have been my anchor. In addition to it, I want to thank my family — my parents Helga and Friedrich as well as my sister Stefanie — for their unconditional love and support. They always believed in me.

Furthermore, I would also like to express my gratitude towards the DFG (Deutsche Forschungsgemeindschaft) for their financial support, without much of my research would not have been possible.

I cannot conclude without conveying my heartfelt appreciation to the idyllic city of Passau. Its charm, beauty, and warmth have made it not just my academic home but also the most wonderful and enjoyable city in the world.

Contents

Chapter 1 Introduction

Thinking is not the ability to manipulate language;
it's the ability to manipulate concepts.

Leslie Lamport

Fault tolerance is a label for all means that prevent a service from failing even when there are faults present in the system (Avižienis et al. 2004). For instance, faults can be compensated by having sufficient redundancy. State machine replication (SMR) is a classical approach for achieving fault tolerance by coordinating all client interactions with a set of n independent server replicas (Schneider 1990). In the Byzantine fault model (Lamport, Shostak, et al. 1982), a threshold t of replicas are considered *faulty* and may behave arbitrarily and even collude as if being orchestrated by an attacker. This makes the Byzantine fault model an appealing design choice for replicated systems in *adverse environments*, in which no replica can be solely trusted, but trust can be still put into a sufficiently large number of replicas.

A seminal work in this field has been the introduction of Practical Byzantine Fault-Tolerance (PBFT) (Castro and Liskov 1999), which can be arguably seen as the first practical solution for a Byzantine fault-tolerant (BFT) SMR protocol for the following reasons: It operates in a partially synchronous (Dwork et al. 1988) network model, operates under an an optimal resilience threshold $t < \frac{n}{3}$ (Lamport, Shostak, et al. 1982), and incorporates several optimizations that make it reach performance comparable to a non-replicated system.

In the last years, BFT SMR research has gained more and more traction because these protocols can be used within blockchain systems as an alternative for Proof-of-Work (Nakamoto 2008) to increase performance despite limitations in respect to their lacking scalability in the number of replicas (Vukolić 2015). For instance, the Hyperledger Fabric (HLF) (Androulaki et al. 2018) platform has been adopted to incorporate the BFT-SMaRt (Bessani, Sousa, et al. 2014) library as an ordering service to achieve high-performance and resilient service execution (Sousa, Bessani, and Vukolić 2018). Moreover, further improvements can be made at the protocol level to make consensus in distributed ledger technology (DLT) *more practical*. For example, weight-enabled active replication (WHEAT) (Sousa and Bessani 2015) is an optimization that decreases latencies in a geographically dispersed environment, and thus can speed up the transaction ordering in HLF when deployed in a wide-area network environment (Sousa, Bessani, and Vukolić 2018).

Generally, the use of BFT SMR in DLT highlights the *scalability aspect* as a novel challenge. Here, research works less frequently address latency improvements (Sousa and Bessani 2015; Bonniot et al. 2020; Eischer and Distler 2021) than throughput improvements (Gilad et al. 2017; Yin et al. 2019; Crain et al. 2021; Neiheiser, Matos, et al. 2021; Cason et al. 2021; Alqahtani and Demirbas 2021). Moreover, DLT makes it necessary to consider deployments to be of a planetary scale, and thus the geographic dispersion of replicas is a further cornerstone that optimizations can target, a research path that has been initiated by works like (Sousa and Bessani 2015; Eischer and Distler 2021). A further challenge is that BFT SMR systems are deployed in more adverse environments, where these systems need mechanisms to react to perceived changes in their environment when striving for optimal performance.

Goal

The high-level goal of this thesis is improving BFT SMR for planetary-scale systems in two properties: Making it *fast*, which means lowering the protocol latency observed by clients, and *adaptive*, which refers to the capability of replicas to react to perceived changes, like network conditions or faulty replicas. In the next subsection, we explain the scope and ambitions this thesis makes in more detail by outlining the research questions that derive from this high-level goal.

1.1 Research Questions

We aim to make BFT SMR more applicable in planetary-scale systems, by both improving its latency through protocol-level optimization and by allowing the protocol to adapt to environmental changes. For this purpose, the thesis addresses the following research questions:

> **Q1** How to serve *fast read-only* (consensus-free) operations correctly in BFT SMR systems?

Read-only operations are a special type of optimistic two-step operation, in which replicas directly respond to clients, thus skipping the agreement pattern. This quick communication pattern can significantly lower request latencies and makes the optimization especially relevant in planetary-scale deployments, in which running agreement is costly (time-wise). However, being first introduced by PBFT (Castro and Liskov 1999), this optimization can endanger the liveness of the protocol if a malicious leader conducts an attack. With the attack, the leader can block operations from correct clients forever, notably, this also affects standard, agreement-based operations (as we show in this thesis).

The description of PBFT does not deliver a formal proof for liveness, and in particular, it does not capture this scenario, because proofs have been described based on the standard, non-optimized protocol description (Castro and Liskov 1999; Castro 2000). Further, design choices made by PBFT have influenced many other BFT SMR protocols that followed, such as HQ (Cowling et al. 2006), PBFT-CS (Wester et al. 2009) and BFT-SMaRt (Bessani, Sousa, et al. 2014). We investigated on the question wether the identified problem can be fixed in a generic way so that the solution can be applied to PBFT and other BFT protocols with similar blueprint like BFT-SMaRt. A solution to this problem should enable the optimization in seminal BFT SMR frameworks, so that read operations can be served fast while still preserving both liveness and linearizability. Moreover, for two suggested solutions that we implemented, we also evaluated the performance in both LAN and WAN settings.

> **Q2** How to improve the *geographic scalability* of BFT SMR by making it adaptive towards its deployed *environment* (i.e., network characteristics of replicas)?

WHEAT is a weighted BFT SMR protocol that demonstrated how to achieve latency improvements through protocol-level optimization, in which higher voting weights are assigned to faster replicas (Sousa and Bessani 2015). Yet, it depends on knowing some optimal configuration a priori and assuming it will never change. A particular challenge is how to utilize the geographic dispersion of replicas to optimize a system during its *runtime*, where this thesis brings up the question of how WHEAT consensus can be made more viable for practical deployment.

We experienced that equipping a BFT system with a self-optimizing mechanism introduces quite novel challenges, the first of them being *self-monitoring* in BFT SMR. Here, we need to define suitable strategies for self-monitoring, which includes the question of how to cope with incomplete or possibly counterfeit measurements from Byzantine replicas. A further challenge is to find an optimal system configuration. While a related BFT work approached a similar question through redundant executions (Aublin, Mokhtar, et al. 2013), we study it with the aid of a prediction model. For this, we need to compute the expected latency benefits for WHEAT consensuses and reason about consensus latency prediction and problems related to assessing configurations.

Furthermore, the overall algorithm should implement safe and deterministic reconfigurations (Lamport, Malkhi, et al. 2010). Moreover, we need to consider the influence of faulty replicas within this scheme: Can we redistribute high voting weights of potentially unavailable (e.g., crashed) replicas, and which threats might malicious replicas impose? Last but not least, another challenge consists in making self-adaption in larger systems more efficient. Here, we need to address a fast-growing system configuration space, i.e., the problem of choosing the replicas that should be assigned a high voting weight. These questions require a careful evaluation of a proposed solution, including the conduction of experiments that validate and show the runtime behavior of the deployed system at planetary scale.

Q3 How to improve the *geographic scalability* of BFT SMR by making it adaptive towards its *perceived threat level* (i.e., managing the resilience threshold used for consensus quorums)?

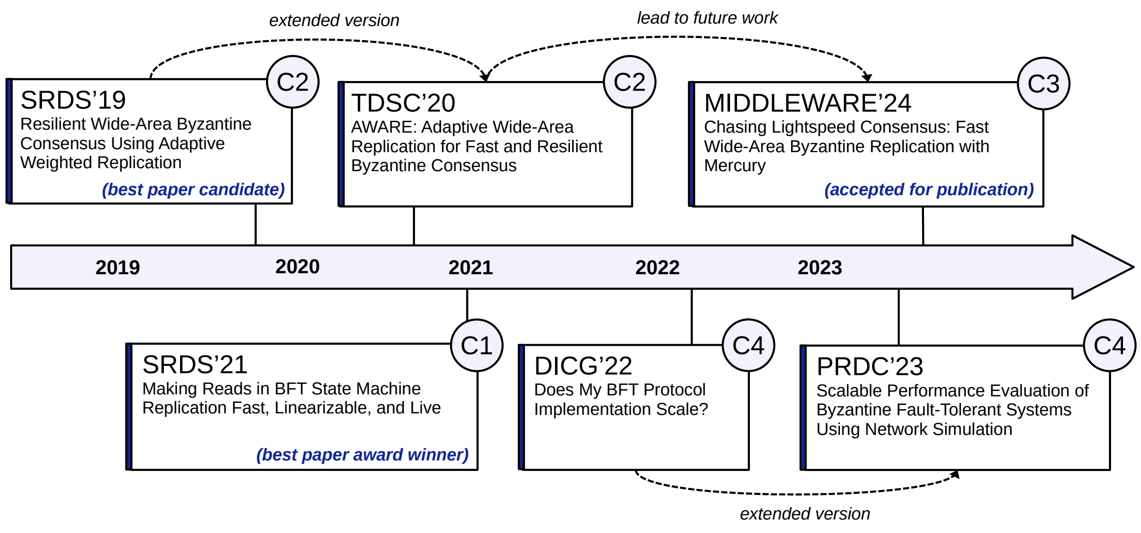
While we worked on AWARE (Berger, Reiser, Sousa, et al. 2022), we found a major hurdle still exists in the choice of whether replicas are used to improve performance or contribute towards the resilience threshold — a design decision, which the system maintainer needed to administer prior to the system's lifespan. The AWARE protocol could drastically improve performance, but in doing so one would sacrifice a not insubstantial degree of resilience, *forever*. In conclusion, AWARE left a significant question untouched: How can we successfully master the inherent trade-off between performance and resilience in such systems?

This question lead to the design of a system that tries to achieve the best of both worlds by autonomously switching between two different modes of operation: a *fast mode*, with a lower resilience threshold, and a *resilient mode*. Concrete research questions target how this design can preserve traditional SMR guarantees, liveness and linearizability, *at all times*, and, under optimal resilience. Since the fast mode uses an underestimated resilience threshold, the BFT SMR protocol needs to be provisioned with suitable detection and repair capabilities that allow it to react in scenarios in which equivocations can occur (Sheng et al. 2021), and then resume in the resilient mode, which makes it altogether a challenging endeavor.

Another challenge insists on lowering the observed *request latencies* even further by making use of *correctable* client-side speculation (Guerraoui, Pavlovic, et al. 2016). The underlining concepts of FlashConsensus demanded thorough evaluations in different network environments. For instance, our evaluations study the speedup that can be achieved by combining the effect of fast consensus quorums with client-side speculation as measured by clients distributed across the planet. A further question is to what extent the general concept of FlashConsensus can be successfully applied to other BFT protocols such as HotStuff (Yin et al. 2019), which uses even more communication steps in its agreement pattern than BFT-SMaRt.

Q4 Can we reason about *performance* of BFT SMR systems at *larger scale* using simulations?

We are curious if simulation can be a suitable and cheap method to investigate the performance of BFT SMR protocol implementations. Fur this purpose we evaluate a concrete method that is based on plugging existing BFT SMR implementations into a high-performance, simulated network environment using Phantom (Jansen et al. 2022). Here, we study the accuracy of simulation results by comparing them with real-world measurements and also reason about resource consumption.



extended version
lead to future work
SRDS'19
Resilient Wide-Area Byzantine Consensus Using Adaptive Weighted Replication
(best paper candidate)
C2
TDSC'20
AWARE: Adaptive Wide-Area Replication for Fast and Resilient Byzantine Consensus
C2
MIDDLEWARE'24
Chasing Lightspeed Consensus: Fast Wide-Area Byzantine Replication with Mercury
(accepted for publication)
C3
2019
2020
2021
2022
2023
SRDS'21
Making Reads in BFT State Machine Replication Fast, Linearizable, and Live
(best paper award winner)
C1
DICG'22
Does My BFT Protocol Implementation Scale?
C4
PRDC'23
Scalable Performance Evaluation of Byzantine Fault-Tolerant Systems Using Network Simulation
C4
extended version

Figure 1.1: A timeline of papers associated with contributions.

1.2 Main Contributions

In summary, this thesis makes the following contributions to address the open research questions:

C1 We discovered that the read-only optimization which accelerates operations by skipping consensus endangers the liveness of seminal BFT SMR protocols, such as PBFT and BFT-SMaRt. We showed an attack and then presented a patch that preserves both liveness and linearizability.

PBFT is a seminal SMR protocol that achieves performance comparable to non-replicated systems in realistic environments. A reason for such high performance is the set of optimizations PBFT introduced. One of these is *read-only requests*, a particular type of client request which avoids running the three-step agreement protocol and allows replicas to respond directly, thus *reducing the latency of reads from five to two communication steps* (Castro and Liskov 1999). Given PBFT's broad influence, its design and optimizations influenced many BFT protocols and systems that followed, e.g., BFT-SMaRt.

For the first time, we show in this thesis that the read-only request optimization introduced in PBFT more than 20 years ago *can violate its liveness*. Notably, the problem affects not only the optimized read-only operations but also standard, totally-ordered operations. Moreover, we also demonstrate this weakness by presenting an attack (the *isolating-leader* attack) in which a malicious leader blocks correct clients. Further, we present two solutions for patching the protocol, thus making read-only operations both fast and correct. These two solutions are implemented on top of BFT-SMaRt and ensure that decisions made in the agreement phase eventually propagate to all correct replicas. We also provide an experimental comparison of the implemented solutions when under attack.

C2 We present Adaptive Wide-Area Replication (AWARE), which enhances the geographical scalability of consensus if replicas are spread out globally. It fuses reliable self-monitoring with a consensus latency prediction model, allowing it to adapt towards its environment through reconfiguration, thus continuously striving for latency gains at runtime.

In BFT SMR systems that are spread out geographically, the speed at which consensus is achieved is fundamentally limited by the heterogeneous latencies of connections between replicas since they need to form quorums. The WHEAT protocol has shown that deployments of planetary scale can benefit from *weighted replication*, a method that uses redundant replicas and assigns higher weights to the fastest subset of replicas (Sousa and Bessani 2015). Notably, the approach allows for more choice in quorum formation and replicas can exploit proportionally smaller quorums to advance protocol stages, thus accelerating consensus decisions. When using weighted replication, the system needs a solution to autonomously optimize itself for its environment (i.e., finding the best distribution of weights which is not trivial). Further, the system should react to changing network conditions or faults.

AWARE improves the geographic scalability of consensus in planetary-scale systems by automatically choosing a weight distribution and leader position that optimizes the system performance given the location of all replicas. Thus, it benefits the emergence of fast quorums in the system. In particular, AWARE follows this idea: Within intervals of consensus instances, replicas repeatedly use a reliable self-monitoring procedure to collect and disseminate measurements that are synchronized between all correct replicas. These measurements are fed into a deterministic consensus latency prediction model to determine the best system configuration for a later reconfiguration.

Furthermore, we employ experiments using several AWS EC2 regions, to validate that AWARE dynamically optimizes consensus latency by self-reliantly finding a fast system configuration and thus also leads to latency gains observed by clients located in different regions of the world. These experiments include a use case with Hyperledger Fabric using AWARE as a consensus substrate to order transactions, i.e., achieving agreement on which block is appended next to the blockchain.

C3 We explain how FlashConsensus refines AWARE by making the resilience threshold adaptive. It achieves awareness about the needed degree of resilience (i.e., the *threat level*) and tentatively uses a lower resilience threshold, that enables compacter quorums and thus accelerates consensus in common scenarios with only few faulty replicas.

An open problem in AWARE has been mastering the resilience-performance trade-off. A key observation that we made is the following: A lower resilience threshold t leads to smaller consensus quorums in the planetary-scale system and thus yields faster performance. However, because t had to be selected prior to the system's deployment and then could not be changed anymore during the system's runtime, it resulted in an inherent trade-off between resilience and performance.

The performance gain derives from the fact that with a smaller threshold, we can obtain smaller consensus quorums and these smaller quorums can consist of only replicas that are closer to each other, and then lead to exchanging messages faster. To achieve the best of both worlds, FlashConsensus follows this idea: By tentatively underestimating the resilience threshold, thus using a lower threshold $(t_{fast}) < t$ than optimal, the system can make use of the smaller and thus faster quorums. This is an optimization for expected scenarios in which there are either none or only a few faulty replicas.

To deal with all the problems that might arise through the tentative use of an underestimated (and possibly *violated*) resilience threshold, the BFT SMR protocol needs to be enriched with detection and self-repair facilities. For this purpose, it is necessary to incorporate mechanisms that allow us to achieve awareness of the threat level, for which BFT protocol forensic support (Sheng et al. 2021) is used, and two operational modes: *fast* and *conservative*. This allows us to retreat from an optimized execution and go back to a resilient protocol execution, in cases where we actually need optimal resilience to maintain the correctness of the system. Remarkably, FlashConsensus always guarantees liveness and linearizability under optimal resilience ($t < \frac{n}{3}$) even when the small quorums are based on a lower resilience threshold.

Moreover, FlashConsensus employs a form of client-side speculation by utilizing incremental consistency guarantees which are finally implemented through a programming abstraction called *Correctable* (Guerraoui, Pavlovic, et al. 2016), which is managed and accessed by the client. Our experimental evaluation considering up to 51 replicas scattered across the planet, shows that FlashConsensus is capable of ordering transactions with finality in less than 0.4 s, which is half of the time required for a PBFT-like protocol in the same network environment and is less than this protocol running on the theoretically best possible internet links (transmitting at 67% of the speed of light) given the location of the replicas.

C4 We examined how well simulation can be utilized as a method to analyze the performance of unmodified BFT SMR implementations in large-scale deployment scenarios.

Experiments with actual protocol deployments usually offer the best realism, but they are costly and time-consuming. We explored a method to simulate unmodified BFT protocol implementations by plugging the implementations into the high-performance network simulator Phantom (Jansen et al. 2022). We discovered that we could faithfully forecast the performance of BFT protocols if performance eventually becomes network-bound at a larger scale. Our evaluations compare results obtained from simulations with measurements of real protocol deployments, indicating that we can accurately forecast the performance of BFT protocols while experimentally scaling their environment.

1.3 Publications

During my work as a PhD candidate, I co-authored several publications. These papers have been published or submitted as manuscripts to peer-reviewed workshops, conferences, or journals.

Parts of this thesis. In particular, parts of this thesis have already been published in the following publications (also shown in the timeline, see Figure 1.1):

- Christian Berger, Hans P. Reiser, João Sousa, and Alysson Bessani (2019). "Resilient Wide-Area Byzantine Consensus Using Adaptive Weighted Replication". In: *Proceedings of the 38th IEEE Symposium on Reliable Distributed Systems (SRDS).*
- Christian Berger, Hans P. Reiser, João Sousa, and Alysson Bessani (2022). "AWARE: Adaptive Wide-Area Replication for Fast and Resilient Byzantine Consensus". In: *IEEE Transactions on Dependable and Secure Computing* 19.3, pp. 1605–1620. DOI: `10.1109/TDSC.2020.3030605`.
- Christian Berger, Hans P. Reiser, and Alysson Bessani (2021). "Making Reads in BFT State Machine Replication Fast, Linearizable, and Live". In: *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pp. 1–12. DOI: `10.1109/SRDS53918.2021.00010`.
- Christian Berger, Sadok Ben Toumia, and Hans P. Reiser (2022). "Does My BFT Protocol Implementation Scale?" In: The 3rd International Workshop on Distributed Infrastructure for Common Good (DICG). Quebec, Quebec City, Canada: Association for Computing Machinery, pp. 19–24. ISBN: 9781450399289. DOI: `10.1145/3565383.3566109`. URL: `https://doi.org/10.1145/3565383.3566109`.
- Christian Berger, Lívio Rodrigues, Hans P. Reiser, Vinicius Cogo, and Alysson Bessani (2024). "Chasing Lightspeed Consensus: Fast Wide-Area Byzantine Replication with Mercury". In: *The 25th ACM/IFIP International Middleware Conference.* ACM.
- Christian Berger, Sadok Ben Toumia, and Hans P. Reiser (2023). "Scalable Performance Evaluation of Byzantine Fault-Tolerant Systems Using Network Simulation". In: *The 28th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC).* IEEE.

Other papers I contributed to. Further, I gladly contributed to a variety of other research papers, though they are not featured in this thesis:

- Christian Berger and Hans P. Reiser (2018a). “Scaling Byzantine Consensus: A Broad Analysis”. In: *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers.*
- Christian Berger and Hans P. Reiser (2018b). “WebBFT: Byzantine fault tolerance for resilient interactive web applications”. In: *Distributed Applications and Interoperable Systems: 18th IFIP WG 6.1 International Conference, DAIS 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings 18.* Springer, pp. 1–17.
- Jörg Domaschka, Christian Berger, Hans P. Reiser, Philipp Eichhammer, Frank Griesinger, Jakob Pietron, Matthias Tichy, Franz J. Hauck, and Gerhard Habiger (2019). “SORRIR: A Resilient Self-Organizing Middleware for IoT Applications [Position Paper]”. In: The 6th International Workshop on Middleware and Applications for the Internet of Things (M4IoT). Davis, CA, USA: Association for Computing Machinery, pp. 13–16. ISBN: 9781450370288. DOI: 10.1145/3366610.3368098. URL: https://doi.org/10.1145/3366610.3368098.
- Christian Berger, Philipp Eichhammer, Hans P. Reiser, Jörg Domaschka, Franz J. Hauck, and Gerhard Habiger (Sept. 2021). “A Survey on Resilience in the IoT: Taxonomy, Classification, and Discussion of Resilience Mechanisms”. In: *ACM Computing Surveys* 54.7. ISSN: 0360-0300. DOI: 10.1145/3462513. URL: https://doi.org/10.1145/3462513.
- Sadok Ben Toumia, Christian Berger, and Hans P. Reiser (2022). “An Evaluation of Blockchain Application Requirements and Their Satisfaction in Hyperledger Fabric”. In: *Distributed Applications and Interoperable Systems.* Ed. by David Eyers and Spyros Voulgaris. Cham: Springer International Publishing, pp. 3–20. ISBN: 978-3-031-16092-9.
- Christian Berger, Hans P. Reiser, Franz J Hauck, Florian Held, and Jörg Domaschka (Sept. 2022). “Automatic Integration of BFT State-Machine Replication into IoT Systems”. In: *2022 18th European Dependable Computing Conference (EDCC).* Los Alamitos, CA, USA: IEEE Computer Society, pp. 1–8. DOI: 10.1109/EDCC57035.2022.00013. URL: https://doi.ieeecomputersociety.org/10.1109/EDCC57035.2022.00013.
- Christian Berger, Signe Schwarz-Rüsch, Arne Vogel, Kai Bleeke, Leander Jehl, Hans P. Reiser, and Rüdiger Kapitza (2023). “SoK: Scalability Techniques for BFT Consensus”. In: *IEEE International Conference on Blockchain and Cryptocurrency (ICBC).* IEEE.

1.4 Organization

This thesis uses Chapter 2 to summarise and explain background knowledge and Chapter 3 to review the state of the art while pinpointing open problems. The advancing Chapters 4-7 present the main contributions of this thesis. Finally, in Chapter 8, we draw conclusions. In more detail, the structure of the remaining part of this thesis is as follows:

- In Chapter 2, we first explain basic terminology and concepts that are relevant to understanding this thesis. For instance, this concerns the introduction of synchrony models, fault models, quorum systems, and broadcast semantics. After that, we recite and explain the consensus problem as well as SMR and review a few famous SMR protocols.
- In Chapter 3, we summarize the state of the art of BFT SMR protocols with respect to (1) optimisations that aim to improve performance in wide-area networks, (2) fast variants of BFT SMR protocols and (3) adaptive BFT SMR protocols. Further, we review the WHEAT protocol in more detail and then outline open problems.
- In Chapter 4, we present a found vulnerability in highly optimized BFT SMR protocols that stems from the integration of fast (two-step) read-only operations. Further, we explain two solutions to patch these protocols to ensure both liveness and linearizability under this optimization and subsequently evaluate our implemented solutions under attack.
- In Chapter 5, we present the AWARE protocol by first explaining its overall approach to self-monitoring and self-optimization. Further, we sketch its implementation in WHEAT and conduct an extensive evaluation within an AWS-based wide-area network. Subsequently, we discuss modifications necessary to improve the scalability of AWARE and evaluate its integration into Hyperledger Fabric as an ordering service.
- In Chapter 6, we present FLASHCONSENSUS, a transformation of quorum-based BFT protocols that can accelerate consensus through a tentative, faster mode of operation in which smaller quorums are used. This is achieved through the judicious integration of BFT forensics techniques and two modes of operation. Subsequently, we discuss the challenge of ensuring liveness and linearizability under optimal resilience even if an underestimated resilience threshold is used. Further, we conduct exhaustive evaluations to reason about the latency gains achievable on a planetary scale.
- In Chapter 7, we present a methodology for conducting simulations of unmodified BFT SMR protocol implementations by plugging these into a high-performance network simulatons. We demonstrate experimentally that simulation can provide high fidelity performance evaluation when the performance of BFT protocols becomes network-bound.
- In Chapter 8, we summarize the main contributions of this thesis, give a few final insights, and sketch possible paths for future work.

Chapter 2
Background

In this chapter, we give a brief summary of relevant notions and concepts that we use throughout this thesis. We start by explaining preliminaries, such as synchrony and fault models, in Section 2.1. In Section 2.2, we review a definition and relevant results for distributed consensus. Finally, we provide an overview of state machine replication fundamentals in Section 2.3.

2.1 Preliminaries and Terminology

To begin with, let us briefly recap some important notions and models for distributed systems, starting with the question of what a *distributed system* actually is. Bal defines a distributed system in the following way:

Definition 2.1.1 **Distributed System**

A distributed computing system consists of multiple autonomous processors that do not share primary memory, but cooperate by sending messages over a communications network (Bal 1990).

Van Steen and Tanenbaum define a distributed system as *a collection of autonomous computing elements that appears to its users as a single coherent system* (Van Steen and Tanenbaum 2017). Core characteristics are the non-existence of a common clock and the non-availability of shared memory — instead, the computing elements collaborate by passing messages to each other over a network.

In the context of this thesis, we understand a distributed system as a collection of *processes* (a process is a running instance of a program) which are distributed on separate computers and communicate over a network to collaboratively achieve a common goal. Communication links are generally assumed to be at least *fair-loss links*, which do not indefinitely drop messages, and which can finally be utilized to implement *perfect point-to-point links* (an abstraction which guarantees that the message of a sender actually arrives at the receiver side) through retransmission attempts in the case of losses (Cachin, Guerraoui, et al. 2011).

Further, a crucial aspect of distributed systems is that the failure of such an individual computing element (for instance, a single process) may have fatal consequences for the whole system. This led Lamport, back in the year 1987, to even call a distributed system as *one in which the failure of a computer you didn't even know existed can render your own computer unusable* (Lamport 1987).

To assert if a distributed system works as intended, it must be first formalized which concrete properties this system should fulfill.

Definition 2.1.2 **Properties of Distributed Systems**

Lamport devised two categories for developing system properties (Lamport 1977):

- *A **liveness** property is one which states that something must happen [eventually].*
- *A **safety** property is one which states that something will not happen.*

To illustrate this, let us assume a traffic light circuit. A liveness property could be that a traffic light eventually signals green, thus allowing cars to bypass a junction. The safety property would state that intersecting traffic lights must never signal green at the same time to avoid cars colliding.

Fault tolerance is a means to attain dependability and security in a distributed system, in particular, Avižienis et al. define fault tolerance as *all means to avoid service failures in the presence of faults* (Avižienis et al. 2004), this includes, for instance, *compensation*, which employs redundancy to mask the occurrence of errors. For instance, *state machine replication*, which we formally define in a later subsection, is a technique that categorizes as compensation. Another technique is a *rollback*, which eliminates the error by transforming the system back to an errorless state which was, e.g., saved prior to the occurrence of the error (often referred to as a *checkpoint*) (Avižienis et al. 2004).

2.1.1 Synchrony Models

Synchrony models help us to reason about temporal behavior and the conceptual understanding of time in a distributed system. For instance, this concerns assumptions we make about the time that passes for a message to be transferred from a sender to the receiver over the network, or the time needed by some process to perform a local computation. In this subsection, we subsume popular synchrony models and discuss with what thoughts in mind they are employed by practical systems.

Synchronous System Model

The synchronous system model assumes that a concept of physical time exists and that strict and plausible assumptions can be made *and always hold* about the timeliness of events. In particular, the synchronous system model assumes the existence of known upper bounds on the time needed for both message transmissions over the network and local computations:

> **Definition 2.1.3** **Synchronous System Model**
>
> *In the synchronous system model, some known upper bound δ exists for all message transmission delays over the network. The same concept also applies to all local computations.*

The benefit of the synchronous system model is that it crucially simplifies the design of fault-tolerant protocols in this model. The parameter δ allows to design safe protocols, e.g., by waiting for (some multiple of) δ before proceeding with the next protocol step. It can be safely assumed that every *correct* process would have participated within the time span. However, in the real world, protocols, which safety depends on δ might be hard to manage. Especially, if choosing δ too small can violate real-world circumstances and lead to unsafe system behavior. And choosing δ too high can lead to unnecessary long waiting intervals that might tarnish system performance.

Asynchronous System Model

On the other hand, there is the asynchronous system model. In this model, timing assumption are not employed. In particular, there is no assumption made about the existence of possible upper bounds on the time needed for network transmissions or performing local computations. In case we say *something happens eventually*, we mean that something happens after some unknown (but *finite*) amount of time.

> **Definition 2.1.4** **Asynchronous System Model**
>
> *In the asynchronous system model, all message transmissions over the network as well as all local computations can be delayed by any (unbounded but finite) amount of time.*

The main benefit of the asynchronous system model is that designed protocols are guaranteed to always work under real (potentially uncivil) network conditions. However, designing correct and

System model	Upper bound δ exists?	Upper bound holds ..
synchronous	yes	always
partially[1]synchronous	yes	after unknown GST
asynchronous	no	–

Table 2.1: Simplified overview over different synchrony models.

fault-tolerant protocols for some problems in this type of model may be more difficult or even impossible. For instance, it is impossible to deterministically achieve consensus in the presence of faults in an asynchronous system (Fischer et al. 1985) while it is possible in a synchronous system (Pease et al. 1980).

We see that both the synchronous and asynchronous system model have their shortcomings: On the one hand, the synchronous model, while simple to use, might not accurately enough reflect reality where timing assumptions could be violated (at least *sometimes*). On the other hand, the asynchronous system model provides generality, but some problems can not be easily solved (or are even impossible to solve) without any timing assumptions being made.

Partially Synchronous System Model

Dwork et al. originally introduced the concept of *partial synchrony* which lies between the cases of a synchronous system and an asynchronous system (Dwork et al. 1988). Partial synchrony is described with two different versions, (1) the *unknown bounds* partial synchrony, where fixed but a priori unknown bounds for message delay and computation delay exist, and, (2) the *global stabilization time (GST)* partial synchrony, where bounds are known to a protocol designer, but only hold *eventually*, i.e., after some *unknown time span* which is modeled by the GST.

For the remainder of this thesis we employ the GST definition of partial synchrony, but we may call this model also an *eventually synchronous system* since, informally speaking, the system behaves just like a synchronous system as soon as the GST is reached.

Definition 2.1.5 **Partially Synchronous System Model**

In the partially synchronous system model, some upper bound δ exists for all message transmission delays over the network which holds after some unknown global stabilization time (GST). The same concept also applies to delays of all local computations (Dwork et al. 1988).

Protocols designed for the asynchronous system also work in a partially synchronous system and synchronous system, because every system is *at least* asynchronous and some systems might behave (partially) synchronously. Thus, the subset relation between these systems is

$$\textsc{Synchrony} \subseteq \textsc{Partial Synchrony} \subseteq \textsc{Asynchrony}$$

In practice, the partially synchronous system model captures a sweet spot between the asynchronous and synchronous models. For instance, it allows a protocol designer to use timers in the protocol description which, i.e., can be incremented to eventually meet the unknown bound. Also, the protocol designer decides how a protocol behaves during phases of asynchrony (either GST has not been reached yet or the unknown δ was underestimated). For instance, BFT SMR protocols like PBFT tend to favor safety (consistency) over liveness (progress) in this scenario (Castro and Liskov 1999).

In respect to real-world systems, the partial synchrony model allows building on the idea that even large networks (like inter-cloud networks, connected through the Internet) behave predominantly synchronously, at least *often enough*. Potential network disturbances are still covered by the model

[1]Also called *eventually* synchronous system model.

as the system is allowed to behave asynchronously at times so long as it eventually recovers its synchronous behavior. For a quick comparison, we give a simplified[2] summary of the different synchrony models in Table 2.1.

2.1.2 Fault Models

In this section, we briefly discuss terminology related to faults, as well as a few different models used in distributed systems to capture their concrete behavior.

Fault, Error, and Failure

In this thesis, we employ the established terminology on fault, error, and failure as well as the concept of a *fault→error→failure* chain as proposed by Avižienis et al. (Avižienis et al. 2004):

> **Definition 2.1.6** **Faults, Errors and Failures**
>
> ***Faults** can be categorized into dormant or active and are the cause for an error, which manifests upon fault activation. The **error** itself is a form of an incorrect state. Errors can lead to further errors, which is called error propagation. A **failure** is the deviation of an erroneous system's behavior from correct system behavior, such as when producing some wrong result due to the erroneous state.*

It is noteworthy that the *fault→error→failure* chain can propagate recursively because the failure of some component C_1 may appear as an external fault to another component C_2 if the correct execution of C_2 depends on receiving correct results from C_1 beforehand. For more detailed taxonomies of these terms, we refer to the extensive survey of Avižienis et al. (Avižienis et al. 2004).

Fault-tolerance allows a service to work as intended (i.e., *preventing service failure*) despite the presence of faults in the system (Avižienis et al. 2004). A system entity is considered as either *correct* or *faulty* at a time. In the scope of this thesis, we assume *processes* to be these system entities and thus need to model the behavior of *faulty processes*.

Fault models allow us to summarize assumptions about the *behavior of faulty processes within a distributed system* which helps protocol designers to come up with solutions on how to deal with them. For instance, these questions quickly come to mind:

- What exact faults do we even consider? It is very important to argue about the concrete behavior faulty processes may display – some actual faults might not be covered by our assumptions.
- Is a faulty process considered faulty forever or can it recover? The temporality of faults is an important detail to consider for practical use.

Moreover, models can also try to capture the *security* aspect and thus assume an evil *adversary* that can orchestrate and control faulty processes. In this case, the extent of power an adversary has must be reasonably explained. Avižienis et al. coined the term *fault assumption coverage* as a measure for reasoning about how closely assumptions of a fault model cover reality (Avižienis et al. 2004).

A very common assumption is that processes may fail *independently*. In practice, this assumption may be easily violated if, for instance, computers are attached to the same power source, or, the operating system or software has shared vulnerabilities that may affect all processes. In the next subsections, we recite and explain some well-established fault models.

[2]It is simplified, because two different bounds can be used to model transmission delay and computation delay.

Crash-Stop Faults

A simple and benign fault model is to only assume *crashes*, i.e., processes fail only by crashing. The crash makes the process stop executing further computation steps or sending (or receiving) messages onward.

> **Definition 2.1.7** **Crash Fault Model**
>
> *A process is* ***faulty*** *if it crashes at some time during its execution, after which it stops executing any local computation and does not send any message to other processes – a* ***correct*** *process never crashes and executes an infinite number of steps (Cachin, Guerraoui, et al. 2011).*

We call this model either *crash* fault model or *crash-stop* fault model, to account for the fact that it does not assume the recovery of processes after they failed.

Omission Faults

Omission faults extend crash faults by assuming that a faulty process might not just simply stop its execution completely (which can be seen as a special case of omitting everything after some point in time), but instead may omit sending or receiving only *individual* messages, e.g., due to buffer overflows or network congestion (Cachin, Guerraoui, et al. 2011), thus adding a bit of complexity to the crash model.

> **Definition 2.1.8** **Omission Fault Model**
>
> *A process is* ***faulty*** *if it does not send (or receive) a message that it is supposed to send (or receive) according to its algorithm – a* ***correct*** *process always sends (or receives) messages it is supposed to send (or receive) (Cachin, Guerraoui, et al. 2011).*

Crash-Recovery Faults

Crash-recovery faults add the possibility for a crashed process to recover later and resume its execution, which naturally makes this model more plausible than the crash-stop model.

In this abstraction, a process can *at any time* stop to send messages due to a crash, but still, recover and resume sending messages later on. Thus, an omission fault can be seen as a special kind of crash-recovery fault, but crash-recovery can further lead to a faulty process suffering from amnesia if it crashes and loses its internal state (Cachin, Guerraoui, et al. 2011).

> **Definition 2.1.9** **Crash-Recovery Fault Model**
>
> *A process is* ***faulty*** *if it either crashes and never recovers or it keeps infinitely often crashing and recovering – a* ***correct*** *process is eventually (after some time) always up and running (as far as the lifetime of the algorithm execution is concerned), which means it crashes and subsequently recovers for only a finite number of times (Cachin, Guerraoui, et al. 2011).*

The crash-stop, omission, and crash-recovery fault models all assume benign (i.e., non-malicious) faults, which seems to be a limitation when we think about the current state of Internet applications. The possibility of an adversary, that, e.g., intrudes into a distributed system and manages to gain control over some processes demands to capture a more sophisticated, possibly malicious behavior of faulty processes, and thus additional fault models are necessary.

Eavesdropping Faults

Eavesdropping faults occur in distributed systems if some *adversary* obtains unauthorized *read access* to information leaked from within a faulty process. Thus, this type of fault affects the

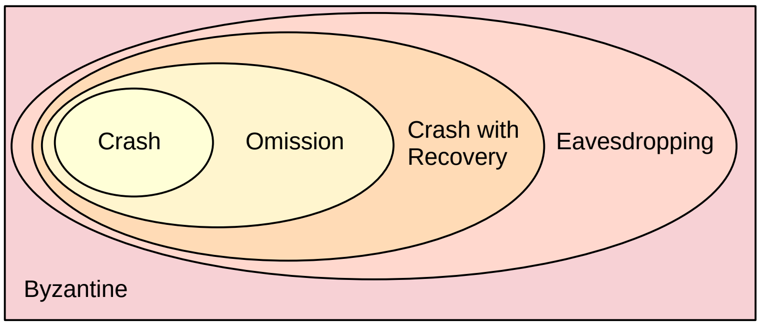


Figure 2.1: Hierachy among different types of faulty processes (Cachin, Guerraoui, et al. 2011).

confidentiality of the system, especially since the adversary might gather pieces of information from multiple faulty processes that are affected.

Definition 2.1.10 **Eavesdropping Fault Model**

*This model extends the crash-recovery model by the assumption that some specific adversary may eavesdrop (obtain information unauthorized) on all **faulty** processes and correlate all leaked pieces of information with each other – **correct** processes do not leak information unintentionally (Cachin, Guerraoui, et al. 2011).*

The malicious behavior of an adversary is still quite limited since he can only observe internals, but not force faulty processes to deviate from their intended protocol behavior in a malicious way (e.g., other than the benign behavior one would assume in the crash-recovery model).

Byzantine Faults

Possibly, the best way to explain Byzantine faults is that these encompass all faults that are possible, and thus are a superset including all of the previously explained faults, *and more*. In literature, Byzantine faults are often explained by assuming an *arbitrary behavior* of faulty processes (Cachin, Guerraoui, et al. 2011). The term *arbitrary* needs to be read with caution because even Byzantine processes might not behave in a manner impossible to them. For example, Byzantine replicas can not communicate with each other faster than the speed of light or break strong cryptographic primitives as their computational power is limited.

Compared to an eavesdropping fault, a Byzantine process is assumed to be under complete control of an adversary, thus the adversary can not only read the internal state but freely alter this state or inject messages into the network, e.g., to fulfill some malicious goal. Note, that *arbitrary* behavior outside of the crash-recovery model is not necessarily the consequence of such an adversary and thus proof of malicious nature, but instead can simply also be just caused by a bug in the implementation of an algorithm (Cachin, Guerraoui, et al. 2011) or caused by hardware defects.

Definition 2.1.11 **Byzantine Fault Model**

*A **faulty** process may display arbitrary (but feasible) behavior. Feasible refers to the behavior being still bounded by (computational, networking, or other) resources or the laws of physics. An adversary is assumed to be in control over **all faulty** processes. A **correct** process is one which always exactly follows its algorithm.*

Through the remaining part of this thesis, we will, unless explicitly stated otherwise, assume the Byzantine fault model. This means that throughout this thesis the terms *Byzantine* and *faulty* are just synonyms. The fault models we recited up to now form a hierarchy as shown by Figure 2.1. In the next subsection, we discuss some capabilities of a hypothetical adversary.

The Threshold Adversary

In the Byzantine fault model, the adversary is assumed to have control over only a threshold t out of n processes in the system. All other $n - t$ processes are correct and show behavior that exactly matches the algorithm. As mentioned earlier, the group of t processes may behave arbitrarily, even displaying colluding behavior, but is still limited by resources and computational feasibility, e.g., the *adversary can not break strong cryptographic primitives*.

Further, the access to information that an adversary has may be *limited*, e.g., through private communication channels between processes, or *unlimited* if the adversary is modeled to have full disclosure on all messages sent over the network (Ben-Or et al. 2006; Abraham, Malkhi, et al. 2019). Finally, the adversary is assumed to be either *static*, i.e., make the selection of t processes at system start without the possibility to change it later on, or *adaptive*, which means the adversary can change the controlled processes as long as this selection never exceeds t at any point in time (Canetti et al. 1999).

Modeling Network Faults

Faults can also occur at the network level, and thus it is necessary to abstract guarantees that the communication system may provide. In the context of this thesis, we follow the notations found in (Cachin, Guerraoui, et al. 2011): We usually employ the primitives of *send(·)* and *deliver(·)* and we consider the transmission of messages over *point-to-point* links through the network. The method *send(·)* is invoked by the sender when initating the sending of a message over a point-to-point link to another process. Furthermore, the event *deliver(·)* at the target process means a message was successfully received by the target process and provides certain guarantees implemented by a communication primitve, for instance, de-duplication or authentication.

Faulty Links. The following types of faults may appear during the transmission of a message over a point-to-point link: message loss, message creation or corruption, and message duplication.

- *Loss*: A message is put into the link by the sender and does not arrive at the target.
- *Duplication*: A message is put into the link by the sender and arrives at the target more than once.
- *Creation*: A message arrives at the target that has not been sent before by the source.
- *Corruption*: A message was altered during the transmission on the link.

In a Byzantine setting, message corruption can also be viewed as a malicious combination of message loss and message creation (i.e., purposely loosing some message and replacing it with a modified message). As a general assumption of this thesis, we assume that a link between two processes may behave faulty sometimes but does not so perpetually. This assumption is described as a *fair-loss point-to-point link*.

Definition 2.1.12 **Fair-loss Point-to-Point Link**

*A link is called a **fair-loss point-to-point link** if it has the following properties (Cachin, Guerraoui, et al. 2011):*

- ***Fair-loss**: If a correct process p infinitely often sends a message m to a correct process q, then q delivers m an infinite number of times.*
- ***Finite duplication**: If a correct process p sends a message m a finite number of times to process q, then m cannot be delivered an infinite number of times by q.*
- ***No creation**: If some process q delivers a message m with sender p, then m was previously sent to q by process p.*

On top of fair-loss links, communication primitives with stronger guarantees can be implemented. For instance, we can deal with lost messages through re-transmission attempts or implement filters for duplicate detection. In a Byzantine fault model, the simple "No creation" property does also not provide sufficient guarantees, since any Byzantine process could forge and insert messages as it pleases. For this purpose, we would additionally need cryptographic primitives to authenticate messages exchanged between two correct processes. An example for a primitive with stronger guarantees that can be built on top of the fair-loss abstraction is the *authenticated perfect point-to-point link*.

Definition 2.1.13 **Authenticated Perfect Point-to-Point Link**

A link is called a ***authenticated perfect point-to-point link*** *if it has the following properties (Cachin, Guerraoui, et al. 2011):*

- ***Reliable delivery****: If a correct process p sends a message m to a correct process q, then q eventually delivers m.*
- ***No duplication****: No message is delivered by a correct process more than once.*
- ***Authenticity****: If some correct process q delivers a message m with sender p and process p is correct, then m was previously sent to q by p.*

In the context of this thesis, we assume that communication links between all processes behave at least fair-loss. The replication library BFT-SMaRt, which we employ for conducting our research in later chapters, establishes authenticated perfect point-to-point links between all system entities through standard network protocols. In particular, it uses TCP which adds reliable delivery and prevents duplication, and TLS, which adds integrity (including data origin authenticity) and confidentiality (an attacker can not read messages exchanged between two correct processes even when having access to their link).

2.1.3 Quorum Systems

In distributed systems, algorithms often use the concept of forming *quorums* for making decisions, in particular with the overall ambition to maintain consistency among a set of servers. A quorum is essentially just a subset of servers that guarantees intersection with all of the other possible quorums. The quorum system is a set of quorums, i.e., it defines which particular quorums are possible and we later review a few examples of constructing quorum systems. Since all quorums pairwise intersect, operations that gathered a single quorum enjoy consistency among all servers.

Because a subset of servers is sufficient to form a quorum, the availability of a system can be increased. In particular, it is possible to construct quorum systems in a way that at least one quorum of correct servers is guaranteed to be available even in presence of t faulty servers.

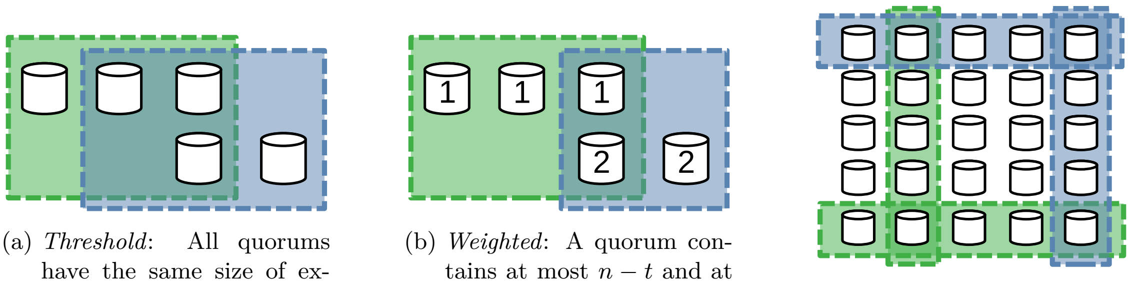
Important Notations

In this subsection, we formally define some important notions in respect to quorum systems by reciting the academic works of Malkhi and Reiter (Malkhi and Reiter 1997).

Definition 2.1.14 **Quorum System and Quorum**

We assume an universe U that encompasses $|U| = n$ servers. A quorum system $\mathcal{Q} \subseteq 2^U$ is a collection of subsets of servers, each pair of which intersect. Intuitively, each quorum $Q \in \mathcal{Q}$ can operate on behalf of the system, thus increasing its availability and performance, while the intersection property guarantees that operations done on distinct quorums preserve consistency (Malkhi and Reiter 1997). This condition can be formally specified as

Q-Consistency: $\forall Q_1, Q_2 \in \mathcal{Q} : Q_1 \cap Q_2 \neq \emptyset$ *(intersection in at least one server)*



(a) *Threshold*: All quorums have the same size of exactly $\lceil \frac{n+t+1}{2} \rceil$ servers.

(b) *Weighted*: A quorum contains at most $n - t$ and at least $2t + 1$ servers.

(c) *Grid*: A quorum is of size $2\sqrt{n} - 1$ (here: $t = 1$).

Figure 2.2: Examples of Byzantine dissemination quorums for $t = 1$ and for different dissemination quorum systems. The green and blue quorums intersect in at least $t + 1$ servers.

The intersection property guarantees consistency because all quorums pairwise intersect. Note, that this definition makes no statement about the availability of quorums in the case of faulty servers. For instance, one could force this intersection by including a single specific server s_0 in all quorums. Thus, for $U = \{s_0, .., s_{n-1}\}$, a quorum system could be $\mathcal{Q} = \{Q \in 2^U : \{s_0\} \subseteq Q\}$. To reason about the properties of quorum systems in scenarios where we expect servers to fail, it is necessary to first introduce the abstraction of a *fail-prone system*.

Definition 2.1.15 **Fail-prone System**

We assume an universe U that encompasses $|U| = n$ servers. A fail-prone system $\mathcal{B} \subseteq 2^U$ is a set of subsets of U, none of which is contained in another, such that some $B \in \mathcal{B}$ contains all the faulty servers. The fail-prone system represents an assumption characterizing the failure scenarios that can occur and could express typical assumptions that up to a threshold of servers fail, as well as less uniform assumptions (Malkhi and Reiter 1997).

The fail-prone system encompasses all assumed failure scenarios, meaning all possible combinations of faulty servers that one is willing to assume. A simple assumption would be to limit these failure scenarios by a fixed threshold $t \leq n$ that represents the maximum number of faulty servers that is assumed and then calculate all possible combinations of these. For instance, for $U = \{s_0, .., s_{n-1}\}$, a fail-prone system, defined through a threshold could be $\mathcal{B} = \{B \in 2^U : |B| = t\}$.

Definition 2.1.16 **Dissemination Quorum System**

A quorum system $\mathcal{Q}$ is called a dissemination quorum system for a fail-prone system $\mathcal{B}$ if the following properties are satisfied (Malkhi and Reiter 1997):

D-Consistency: $\forall Q_1, Q_2 \in \mathcal{Q}, \forall B \in \mathcal{B} : Q_1 \cap Q_2 \nsubseteq B$ *(intersection in at least one correct server)*

D-Availability: $\forall B \in \mathcal{B}, \exists Q \in \mathcal{Q} : B \cap Q = \emptyset$ *(quorum availability despite faulty servers)*

In Byzantine environments, the *dissemination quorum* is suited for services that receive and redistribute self-verifying information such as signed client requests, which Byzantine servers can not undetectably alter (Malkhi and Reiter 1997), and is practically used within many agreement-based BFT protocols like PBFT (Castro and Liskov 1999) or BFT-SMaRt (Bessani, Sousa, et al. 2014). Consistency is ensured by the pairwise intersection in at least one correct server, while availability is ensured by the existence of at least one quorum that is composed of only correct servers.

Constructions of Dissemination Quorum Systems

In the following, we briefly present three constructions of dissemination quorum systems: A simple example that is based on a threshold parameter t, followed by an example that assigns weights to all servers, and finally a construction that is based on the arrangement of servers within a grid. In Figure 2.2, we display how quorums of these dissemination quorum systems would look like.

Threshold. A simple instantiation is based on a fixed threshold t, the *t-dissemination quorum system*. Its construction is possible iff $n > 3t$ and defined by $\mathcal{Q} = \{Q \in 2^U : |Q| = \lceil \frac{n+t+1}{2} \rceil\}$ (Malkhi and Reiter 1997).

Weighed. Suppose a system is composed of $n = 3t + 1 + \Delta$ servers, where the Δ servers are used to improve performance instead of adding to the threshold. Here, a binominal weighting scheme can be applied, where $2t$ servers obtain weight $v_{max} = 1 + \frac{\Delta}{t}$ and all other servers obtain weight 1. The total weight in the system is $3tv_{max} + 1$ and both consistency and availability are ensured if up to t servers fail (Sousa and Bessani 2015). Further, let $w(s)$ denote the weight of a server. The construction of a dissemination quorum system is given by $\mathcal{Q} = \{Q \in 2^U : |Q| \leq n - t \land \sum_{s \in Q} w(s) \geq 2tv_{max} + 1\}$ and the concrete size of every possible quorum in the system varies between $2t + 1$ and $n - t$ servers (Sousa and Bessani 2015).

Grid. Let n be a square number and servers are arranged in a grid with columns and rows of length $\sqrt{n} \times \sqrt{n}$. For a fail-prone system $\mathcal{B} = \{B \in 2^U | \, |B| = t\}$, where $2t + 1 \leq \sqrt{n}$, the dissemination quorum system is constructed by $\mathcal{Q} = \{C_j \bigcup_{i \in I} R_i : I, \{j\} \subseteq \{1, .., \sqrt{n}\}, \ |I| = t + 1\}$ (Malkhi and Reiter 1997).

2.1.4 Broadcast Semantics

In this subsection, we review types of broadcasts and their semantics. The broadcast serves as an important abstraction in distributed systems, which defines guarantees of one-to-all communication between one process that initiates the *broadcast* of some message m (but might fail) and all processes that need to *deliver* m. Using these interfaces, we define guarantees and abstract from the lower-level *sending* and *receiving* of messages through the communication network, for which we assume the existence of perfect point-to-point *ppp* links, over which messages are never lost, but might get reordered.

Best-Effort Broadcast. The best-effort broadcast ensures that, if the process that *broadcasts* m is *correct*, then all other correct processes will eventually *deliver* m. This *broadcast* can be easily implemented by calling `ppp.send(m)` for every process in the system, which in turn, call `deliver(m)` once m was *received*.

Further, there are no promises made about the order of message delivery. Moreover, if the sender is faulty, no guarantees are given about the delivery of m by the other processes, i.e., some processes might *deliver* m while others do not. The problem of a faulty broadcasting process is tackled by the *reliable broadcast*.

Reliable Broadcast. The *reliable broadcast* solves the problem of a faulty broadcasting process. In this scenario, the abstraction guarantees *all-or-nothing* behavior on the message delivery among all correct processes in the system, which either all *deliver* m or nobody delivers m. It also prevents duplication or the creation of delivered messages. We employ the definition of Cachin et al. (Cachin, Guerraoui, et al. 2011):

Definition 2.1.17 **Reliable Broadcast**

A broadcast is reliable iff it fulfills the following properties (Cachin, Guerraoui, et al. 2011):

***RB1: Validity**: If a correct process p broadcasts a message m, then p eventually delivers m.*

***RB2: No duplication**: No message is delivered more than once.*

> ***RB3: No creation****: If a process delivers a message m with sender s, then m was previously broadcast by process s.*
>
> ***RB4: Agreement****: If a message m is delivered by some correct process, then m is eventually delivered by every correct process.*

Implementations of a reliable broadcast can be based on letting processes echo a received message to all others before delivering it (for instance, see the extensive survey of Hadzilacos and Toueg (Hadzilacos and Toueg 1993)). Additionally, in a Byzantine environment, quorums need to be formed to ensure the agreement property (one of the earliest works is the algorithm of Bracha et al. (Bracha 1987)).

The reliable broadcast makes no promises about the order in which messages are delivered. For many applications (and also for replication protocols) guarantees about the order of messages are essential. This is why a reliable broadcast primitive can be employed to build more advanced broadcast primitives which are reliable *and* and additionally give guarantees about the order of message delivery.

FIFO Broadcast and Causal Broadcast. First-in-First-out (FIFO) order refers to maintaining the order of messages defined by the order of calls to the *broadcast* primitive as seen by each broadcasting process. Causal order means that if some message could have caused another message, e.g., it was seen by a process before it broadcasted another message, then this casual relation is maintained by all processes that need to deliver the causing message first. We borrow the definitions of Cachin et al. (Cachin, Guerraoui, et al. 2011):

> **Definition 2.1.18** **FIFO Order Delivery**
>
> *If some process broadcasts message m_1 before it broadcasts message m_2, then no correct process delivers m_2 unless it has already delivered m_1 (Cachin, Guerraoui, et al. 2011).*

> **Definition 2.1.19** **Causal Order Delivery**
>
> *For any message m_1 that potentially caused a message m_2, i.e., $m_1 \rightarrow m_2$, no process delivers m_2 unless it has already delivered m_1 (Cachin, Guerraoui, et al. 2011).*

The FIFO broadcast can be implemented by using a reliable broadcast and then adding sequence numbers to messages so that the delivery of a message m can be delayed until all messages from the same broadcasting process with a lower sequence number than that of m have been delivered (Hadzilacos and Toueg 1993). A simple way to implement the causal broadcast is attaching causally preceding messages to the message that is being broadcast so that these messages can be delivered in causal order by the receiving side (Hadzilacos and Toueg 1993).

> **Definition 2.1.20** **FIFO Order Broadcast**
>
> *Assume a reliable broadcast primitive rbc. Then, rbc is called FIFO order broadcast if it fulfills FIFO order delivery.*

> **Definition 2.1.21** **Causal Order Broadcast**
>
> *Assume a reliable broadcast primitive rbc. If rbc fulfills causal order delivery, then it is called causal order broadcast.*

Total Order Broadcast. Total order delivery is a property that will be important in the later parts of this thesis. This is because state machine replication protocols like PBFT and BFT-SMaRt fulfill total order delivery to determine the execution order of client requests. In particular, they can ensure each replica executes through the same requests in the same order, which finally leads to identical state transitions and preserves the consistency of state among correct replicas.

Definition 2.1.22 **Total Order Delivery**

Let m_1 and m_2 be any two messages and suppose p and q are any two correct processes that deliver m_1 and m_2. If p delivers m_1 before m_2 , then q delivers m_1 before m_2 (Cachin, Guerraoui, et al. 2011).

This property, by transitivity states that all messages will be delivered in the same order, *if they are delivered* (which is not a guarantee of total order delivery).

Definition 2.1.23 **Total Order Broadcast**

Assume a reliable broadcast primitive rbc. Then, rbc is called total order broadcast if it fulfills total order delivery.

An implementation of a total order broadcast can be based on a multi-round Byzantine consensus primitive as a building block. In such an implementation, the broadcast primitive would lead to a new instantiation of some consensus round r, which will be finally decided by all correct processes in r, so that they agree on delivering m as r-th message (for simplification of exposition, here we assume just one message per consensus) and needs a designated sender (also called *leader*) (Hadzilacos and Toueg 1993). The total order broadcast is also often referred to as *atomic broadcast* in academic literature. In the following section, we shed more light on *consensus* and its relation to other fundamental problems in distributed systems, in particular, state machine replication.

2.2 The Consensus Problem

The consensus problem and in particular its solutions are essential because any algorithm that solves consensus (or total order multicast) can serve as a building block for state machine replication (Sousa and Bessani 2012). Although we introduce and stick with a single definition for the remainder of this thesis when we refer to *consensus*, it is still noteworthy that actually different flavors of the consensus problem exist, e.g., see (Garay and Kiayias 2020). In this section, we present a definition of consensus (Chandra and Toueg 1996) and briefly summarize some important theoretical results. We illustrate the problem in Figure 2.3.

2.2.1 Definition of Consensus

In a distributed system consisting of a set of n processes $\mathcal{P} = \{p_1, ..., p_n\}$, the consensus problem can be formulated using an interface with two functions available for each process $p \in \mathcal{P}$: propose(v) and decide(u). The function propose(v) is invoked by a process to start the consensus with some proposal v and decide(u) is called by the consensus algorithm upon completion to deliver the consensus decision u back to the process.

Simply put, the *consensus* problem assumes each process i can propose some value v_i as input, and from the set of proposed values $V = \{v_1, ..., v_n\}$, a single chosen value $u \in V$ must be eventually output by *all correct* processes as their *decision*, i.e., the decide(u) event is triggered *only once* for each process.

Consensus Definition by its Properties

In particular, the consensus problem can be defined through a set of properties which are defined over the primitives that consensus participants possess, i.e., proposing some value and deciding on a value.

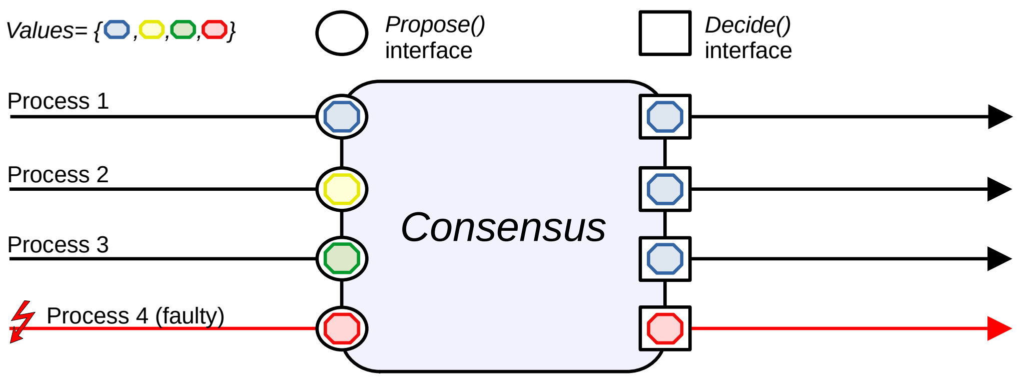


Figure 2.3: Illustration of the consensus problem among four processes.

Definition 2.2.1 **Consensus**

A consensus algorithm, in which processes may propose values $\in V$ and subsequently decide on some common value must satisfy the following properties (Chandra and Toueg 1996):

- ***Termination**: Every correct process eventually decides some value.*
- ***Agreement**: No two correct processes p, p' decide differently:*

$$p \textit{ decide}(v) \wedge p' \textit{ decide}(v') \implies v = v'$$

- ***Uniform Integrity**: Every process decides at most once.*
- ***Uniform Validity**: If a process p decides v, then v was proposed by some process:*

$$p \textit{ decide}(v) \implies \exists p' \in \mathcal{P} : p' \textit{ propose}(v)$$

Any algorithm that can fulfill all the properties above solves the consensus problem. Further, it is noteworthy that there is no requirement that states, that the value v that is decided by all processes must have been proposed by a *correct* process. The validity property is in some definitions modified in particular for its use in the Byzantine fault model. For instance, it may be required that the decided value must have been proposed by a correct process, too:

- **Weak validity**: If all processes are correct and propose v, then a correct process may only decide v. (Cachin 2009)
- **Strong validity**: If all correct processes propose the same value v, then no correct process decides a value different from v; otherwise, a correct process may only decide a value that was proposed by some correct process or the special value $\perp$ (Cachin, Guerraoui, et al. 2011).

2.2.2 Important Results

Over decades of research on the theoretical limitations of consensus in distributed systems, several impossibility results as well as possible solutions to the consensus problem have been made:

Pease, Shostak, and Lamport showed on the example of the Byzantine generals' problem, that Byzantine consensus is impossible in a synchronous system model if $t \geq n/3$ *and* only oral messages can be used (this is also refered to as three generals' problem) but solvable for $t < n/3$ *or* if signed messages can be used (Pease et al. 1980; Lamport, Shostak, et al. 1982). Shortly after that, Ben-Or proposed a probabilistic Byzantine consensus protocol for the asynchronous system model that works if $t < n/5$ (Ben-Or 1983). Later, Bracha showed how to improve this bound for asynchronous Byzantine consensus to $t < n/3$ (Bracha 1987). Further, Dolev and Strong established a lower bound for how many rounds are needed to establish consensus: In the synchronous system model, *authenticated* Byzantine consensus can be achieved in $t + 1$ rounds and with $O(t \cdot n)$ message

Algorithm 1: State Machine Behavior (simplified, compare to (Nayak and Abraham 2019)).

```
state ← $s_0$                                         ▷ Start in initial state
While TRUE do
    waitFor receive ⟨COMMAND, input⟩ from client do
        ⟨output, nextState⟩ ← $exec$(input, state)    ▷ Execute the command
        state ← nextState                             ▷ Update state
        send ⟨REPLY, output⟩ back to the client
```

complexity (Dolev and Strong 1983).

A major breakthrough has been the impossibility result made by Fischer, Lynch, and Patterson: Deterministic, fault-tolerant consensus in the asynchronous system model is impossible, but probabilistic consensus under asynchrony is possible in $O(1)$ expected time (Fischer et al. 1985). Dwork et al. investigated the limitations of Byzantine consensus in the partially synchronous system model: t-resilient Byzantine consensus under partial synchrony is impossible for $t \geq n/3$ but is solvable for $t < n/3$ (Dwork et al. 1988).

Furthermore, Chandra et al. showed that consensus can be solved even with unreliable failure detectors that make an infinite number of mistakes and that atomic broadcast and consensus are bidirectionally reducible in asynchronous systems with crash failures (Chandra and Toueg 1996).

Consensus as a Building Block

The consensus problem is fundamental in distributed systems. Algorithms that solve consensus can be also employed as a building block in modular approaches, e.g., to construct a total order broadcast (Chandra and Toueg 1996), or a state machine replication protocol (Sousa and Bessani 2012). We explain this in more detail in the next section which addresses state machine replication protocols.

2.3 State Machine Replication (SMR)

State machine replication (SMR) is a general technique for achieving fault tolerance by replicating a centralized service on several independent servers, which emulate the service. These are called *replicas*. The SMR protocol provides clients the illusion as if interacting with a single, centralized service, and can mask the failure of a bounded number of replicas.

In this section, we first summarize the basics (e.g., requirements and guarantees) of SMR (Section 2.3.1) and subsequently review the Practical Byzantine Fault-Tolerance (PBFT) algorithm (Section 2.3.2) since its design has been influential for many BFT SMR protocols that followed e.g., (Cowling et al. 2006; Wester et al. 2009; Sousa and Bessani 2012).

2.3.1 Requirements, Guarantees and Programming Model

In SMR, the *state machine* serves as an abstraction for the service implementation that we consider for replication. A state machine *sm* is a tuple $(S, I, O, exec, s_0)$ where S is a set of states, I the set of inputs, also called *commands* which can transition the state and O is a set of possible outputs, the function $exec : (I \times S) \rightarrow (O \times S)$ is called *state transition*, or, *execution*, which maps a pair of command and current state $\langle c, s \rangle$ to a pair of output and next state $\langle o, s' \rangle$ and, finally, the variable s_0 is the distinguished starting state (or *initial* state) of the state machine (Lamport, Malkhi, et al. 2010).

Algorithm 1 shows the basic behavior of a state machine that receives commands from clients, updates its own state, and then responds back with the obtained output. Now, a t-fault-tolerant state machine, or replicated state machine (RSM) is the following abstraction: A set of state machine replicas implement behavior that matches exactly the one of Algorithm 1 (e.g., emulating execution as if only a single machine was involved) but can tolerate up to t replicas being faulty.

Requirements

Schneider explains the following set of requirements for the implementation of the SMR paradigm (Schneider 1990):

Definition 2.3.1 **SMR Requirements**

The requirements of SMR are defined as (Schneider 1990):

- ***Iinitial State**: All correct replicas start in the same initial state s_o.*
- ***Determinism**: Execution is deterministic. This means execution can be modeled as a function over the tuple $(I \times S)$, and thus only ever depends on the received input and current state.*
- ***Coordination**: All correct replicas receive and process the same sequence of requests.*

  ***Agreement**: Every correct sm replica receives every input.*

  ***Order**: Every correct sm replica processes requests it receives in the same relative order.*

Although Schneider states that these requirements are essential for an SMR implementation, he also explains that the notion of these properties can be *sometimes* relaxed. For instance, consider the *coordination* property: we can exploit application-specific knowledge about commands to relax agreement (i.e., *read* commands that do not alter state and do not need to be processed by every replica[3]) or to relax the order of command execution, e.g., if we know that two commands o and o' commute, then their relative order of execution can be safely switched (Schneider 1990).

In practice, the requirements of starting in the same *initial state* and service *determinism* are not as simple as one would think (Bessani and Alchieri 2014). This is because, in a dynamic replication group, a new replica may join during the runtime of the system, and needs to install some correct state that must be transferred to it from the other replicas first. Further, many services use multi-threading or employ data structures that do not have inherent deterministic behavior.

Guarantees

The SMR approach should overall guarantee the following two properties, which can be differentiated as a safety and a liveness criterium according to Castro (Castro 2000):

Definition 2.3.2 **SMR Safety (Linearizability)**

***Safety** means the replicated service behaves like a centralized implementation that executes operations atomically one at a time (Castro 2000).*

Definition 2.3.3 **SMR Liveness (Termination)**

***Liveness** means any operation issued by a correct client eventually completes (Lynch 1996).*

Preserving *linearizablity* (Herlihy and Wing 1990) assuming the non-replicated application also displays linearizable behavior is a *safety* property of SMR which can be used to reason about correct behavior. This means the replicated service should preserve the consistency of a centralized implementation, and not weaken it. For instance, SMR safety ensures that a client never observes behavior from the RSM that deviates from the behavior of a single state machine, e.g., observing an uncommitted state or reading old values. SMR safety is typically achieved by letting all correct

[3]Especially in CFT systems it can be beneficial to read only from a single replica.

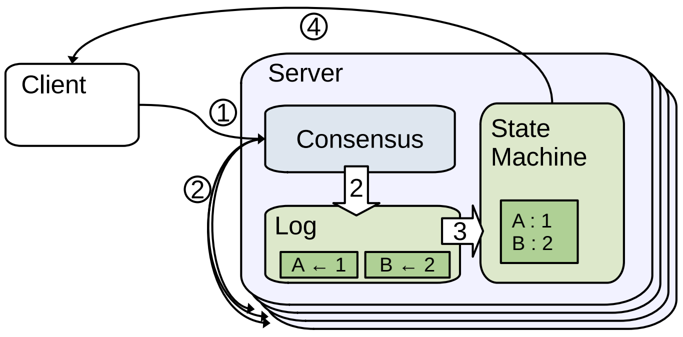


Figure 2.4: Architecture of a RSM using Raft's log-based replication (Ongaro and Ousterhout 2014).

replicas execute the same sequence of operations and enforcing determinism (Bessani and Alchieri 2014).

Termination is the *liveness* property of SMR and guarantees the replicated service's availability. In particular, an operation o *completes* once the client can accept some obtained result of the RSM as the correct result of o. Provided that o has been issued by a *correct* client, then the SMR protocol must be able to ensure that this is *eventually* the case.

Decision log. One commonly employed approach to ensure SMR safety is the use of a consensus protocol that enforces the execution of all operations in total order across all replicas. This results in the creation of a replicated *decision log* abstraction, where each position in the log, denoted as i, can hold at most one finalized operation, or a batch of operations, similar to the way blockchains function.

Safe under asynchrony. As a practical design decision, most SMR protocols in the partially synchronous system model tend to favor safety over liveness, (e.g., see (Castro and Liskov 1999)). These protocols preserve safety at all times, even under asynchrony. Liveness is ensured once the system behaves sufficiently synchronously (typically modeled by a *global stabilization time*). This design is motivated by the thought that during asynchrony, it is mostly better to do nothing and wait (i.e., let the client timeout and retry), rather than allow inconsistencies in the system.

Example: Raft a Crash Fault-Tolerant SMR Protocol

As an example of a concrete implementation serves the Raft's log-based replication (Ongaro and Ousterhout 2014): The SMR paradigm is implemented by an RSM that maintains a consistent log of commands (see Figure 2.4). ① A client provides his command as input to the RSM. ② The consistency of the log is achieved through the Raft consensus algorithm which orders the commands of all clients and ensures they are eventually committed in each RSM.

After commitment, commands are executed by all state machines in the order in which they appear in the log, so that all correct state machines ③ go through the same sequence of state transitions $s_0 \rightarrow \ldots \rightarrow s_k$ and ④ produce the same sequence of outputs $o_0 \rightarrow \ldots \rightarrow o_k$. In particular, for two correct state machines sm and sm', where sm processed k commands, and sm' processed k' commands, holds, that if $k > k'$, then the sequence of state transitions of sm' is a prefix of the state transitions made by sm.

BFT-SMaRt: The Programming Model

We now explained the basic behavior of clients and servers but left the question open how replicated services can be *practically* implemented. For a practical implementation of the SMR paradigm,

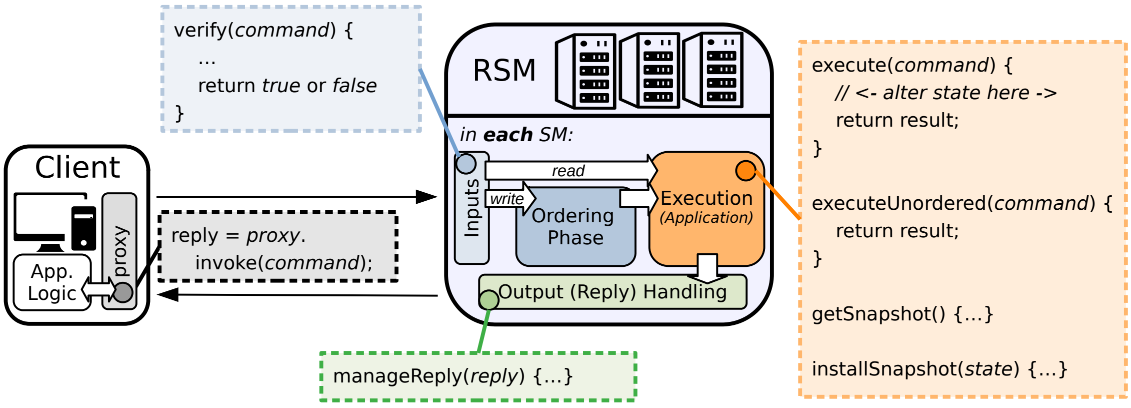


Figure 2.5: Programming interfaces of a BFT-SMaRt RSM.

researchers have crafted several SMR libraries (Castro and Liskov 1999; Clement, Kapritsos, et al. 2009; Bessani, Sousa, et al. 2014). These SMR libraries typically provide interfaces to encapsulate and separate application logic, i.e., functionality that needs to be implemented by the application developer, in a modular way from the underlying SMR protocol implementation.

In the following, we briefly summarize typical SMR programming interfaces by employing the example of the BFT-SMaRt library (Bessani, Sousa, et al. 2014) (for an illustration see Figure 2.5).

Client application interfaces. The client application utilizes an interface called `invoke(`*command*`)` to issue *commands* against the RSM. This `invoke` method is typically implemented by some proxy (that is implemented on the client side, also called service-proxy) which serves as an abstraction: The client application logic can issue its request against this proxy as if it would issue its request against a centralized service. Thus, SMR implementation-level details, such as the client-replicas interaction, remain hidden from the application developer – meanwhile, the proxy may communicate with several replicas for distributing a request or consolidating several matching responses back into a single result that can be delivered to the waiting client application.

A client may also want to mark some commands as read-only so that they skip the ordering phase and can be thus processed faster[4]. This can be implemented by letting the SMR library provide a `invokeUnordered(`*command*`)` method for the client application to use for reading the application state.

Server application interfaces. The server application logic is implemented within the `execute(`*command*`)` interface. The ambition of the `execute` method is to abstract the actual service implementation from the SMR library. The `execute` function is called by the SMR library after some client *command* was successfully ordered and is thus safe to be consumed by the service. The method administrates (reads or writes) the server application state and returns output back to the SMR library which then replies back to the respective client. An important restriction for application developers is, that any service functionality executed in, or, called from within the `execute` function must be deterministic. If read-only requests are used, then the developer may need to implement the handling of read-only requests in a separate `executeUnordered(`*command*`)` method, that is not allowed to alter application state.

SMR libraries typically provide interfaces for creating snapshots of the application state and for installing application state (such as `getSnapshot()` and `installSnapshot(`*state*`)` as shown in Figure 2.5). Interfaces like these are necessary to support the longevity of SMR: for recovery of replicas or system reconfiguration i.e., replicas join during the runtime of the system and receive

[4]This optimization is particularly interesting for read-heavy workloads.

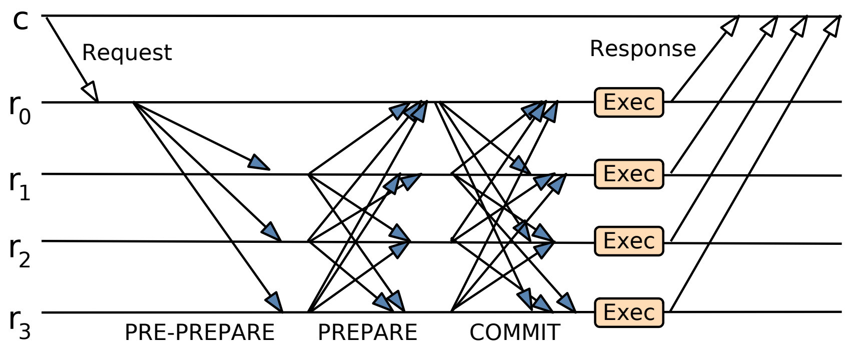


Figure 2.6: Normal case operation of PBFT (Castro and Liskov 1999).

transferred state from other replicas for installation. The application developer can specify how the application state is serialized, and, de-serialized.

As optional features, the interfaces `verify(`*command*`)` and `manageReply(`*reply*`)` help developers to further align the behavior of the RSM with application-specific demands. The function `verify` is used to check if commands are actually valid, meaning in accordance with application semantics. Further, `manageReply` allows developers to build customizable reply management for their applications. It allows replies to be forwarded to clients other than the original sender of the corresponding command. For instance, reply management can be used to create publish-subscribe brokers where a client update is distributed to all subscribers for a topic (managed by the application).

2.3.2 PBFT: Practical Byzantine Fault-Tolerance

In 1999, Castro and Liskov presented the Practical Byzantine Fault Tolerance (PBFT) SMR algorithm, which was well received by researchers as the first practical approach for tolerating Byzantine faults (Castro and Liskov 1999). Provided not more than $t < n/3$ replicas are faulty (which is the optimal resilience bound), PBFT guarantees SMR safety under asynchrony and SMR liveness in a partially synchronous system model. Moreover, PBFT incorporates a set of optimization techniques to achieve throughput comparable to non-replicated services, thus proving its own practicality for real-world applications. At its core, PBFT orders requests by relying on a single leader that associates sequence numbers to requests. As long as this leader remains correct and there is enough synchrony in the system, PBFT's *normal case operation* is repeatedly successfully executed.

Normal Case Operation

PBFT assumes replicas to be organized in *views*, which are basically just a succession of system configurations. The view determines who is a leader and who is a follower: There can never be two leaders in the same view, as the leader is derived in a round-robin fashion by computing the view number modulo n. The view number v is monotonically increasing, which is only triggered by the *view change* sub-protocol, which we save up for later. Moreover, PBFT's voting phases employ Byzantine t-dissemination quorums, so that any set of $2t + 1$ replicas out of a total of $|R| = n = 3t + 1$ replicas form a quorum.

Next, we summarize PBFT as it was explained by Castro and Liskov in the OSDI paper (Castro and Liskov 1999):

Input dissemination. PBFT starts with a client c sending a request $m = \langle \textsc{request}, o, \tau, c\rangle$ to the leader, where o is the operation to be executed and τ a timestamp used to ensure *exactly-once* semantics. Upon a timeout, the client may choose to either retry or broadcast the request to all replicas. The leader of the view v assigns a sequence number s to the request m. Sequence numbers determine the total order of execution (i.e., operation o in m is executed as the s-th operation). Then, the leader writes the request to its message log and broadcasts a $\langle \textsc{pre-prepare}, v, s, m\rangle$[5] to all replicas.

All followers in view v wait for an authenticated $\langle \textsc{pre-prepare}, v, s, m\rangle$ message from the leader and accept it, if they have not yet accepted another PRE-PREPARE before in v and the sequence number s was not used before[6]. Upon acceptance, each follower i broadcasts a $\langle \textsc{prepare}, v, s, h(m), i\rangle$[7] message to all other replicas, where $h(m)$ denotes the hash of m. Followers add both the received PRE-PREPARE and the sent PREPARE messages to their log.

Two-phase commit. All replicas in view v wait for a quorum of authenticated $\langle \textsc{prepare}, v, s, h(m), i\rangle$ messages (including their own PREPARE message) from different replicas $i \in R$ in their log. Here, the PRE-PREPARE of the leader counts as his PREPARE vote. Upon gathering this quorum, each replica i broadcasts a $\langle \textsc{commit}, v, s, h(m), i\rangle$ message to all others and adds the sent COMMIT message to the log. We now say replica i has *prepared* s for m in v.

All replicas in view v wait for a quorum of authenticated $\langle \textsc{commit}, v, s, h(m), i\rangle$ messages from different replicas $i \in R$. Upon gathering this quorum, each replica commits the message m. We now say replica i has *committed* s for m in v. Once m is committed, its contained operation o can be safely *sequentially* executed for its sequence s: Replicas execute o only after all operations for all lower sequence numbers have been executed, therefore ensuring the *total order* property.

Output consolidation. After execution, each replica i responds with the result of execution res back to the client c, by sending a $\langle \textsc{reply}, v, \tau, c, i, res\rangle$ message to it. The client waits for $t+1$ authenticated $\langle \textsc{reply}, v, \tau, c, i, res\rangle$ messages from different replicas $i \in R$ that match in result res and timestamp τ. If that happens, the client *accepts* res for τ. The client can be sure that the obtained result must be correct, because, in any set of $t+1$ replica responses, there is at least one correct replica asserting the correctness.

View-Change

If the leader is correct, then the normal case operation explained before succeeds (under the typical assumptions of no more than t faulty replicas and sufficient synchrony in the system). However, a problem occurs in case of a faulty leader that can only be resolved by the *view-change* which synchronizes information about previous protocol runs among replicas (Castro and Liskov 1999) (see also Figure 2.7):

The view-change protocol is initiated by a replica r in view v once a timer on some request expired. In that case, r discontinues accepting messages apart from VIEW-CHANGE, NEW-VIEW or CHECKPOINT messages and sends a $\langle \textsc{view-change}, v+1, seq, C, P, r\rangle$ to all other replicas. The seq serves as the sequence number for the last stable checkpoint $chkp$ known to r, C is a set of $2t+1$ valid checkpoint messages that proves the correctness of $chkp$ and P is a set which contains so-called *prepare-sets* P_m for each message m that prepared at r with a sequence number higher than seq. Each prepare-set contains $2t$ signed prepare messages along with a valid pre-prepare message with the same view, digest, and sequence number as m. These prepare-sets are used to transfer information about which messages were successfully prepared at replicas in the past to the next leader.

[5] We omit some details for the purpose of a simplified exposition. Because the PRE-PREPARE message later serves as proof and should be kept small, m is actually replaced by its hash $h(m)$ inside the signed PRE-PREPARE and then piggybacked: the leader actually sends a message of the form $\langle\langle \textsc{pre-prepare}, v, s, h(m)\rangle_{\sigma_l}, m\rangle$ where σ_l denotes the signature of the leader.

[6] The sequence also needs to be within a certain allowed range, defined by so-called *watermarks*. These request watermarks allow the leader to start multiple instances of the PBFT normal case operation concurrently.

[7] PREPARE and COMMIT are also signed.

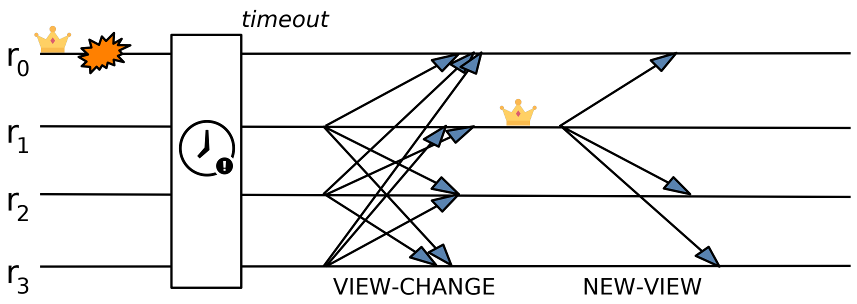


Figure 2.7: View change in PBFT.

Subsequently, the new leader is determined in a round-robin fashion, where the remainder of the modulo division of $v + 1$ by n yields the replica id determined to be the next leader. Upon having collected $2t$ valid VIEW-CHANGE messages from the other replicas, the new leader sends a $\langle$NEW-VIEW$, v + 1, seq, V, O\rangle$ message to all. Here, V is a set of $2t + 1$ VIEW-CHANGE messages (including the one of the leader) to prove the legitimacy of his regency and O is a set of PRE-PREPARE messages, in which the leader (re-)creates PRE-PREPARE messages for all sequence numbers for which he knows a normal case operation started. The PRE-PREPARE messages in O belong to sequence numbers where some messages have been successfully prepared under the last leader (these are re-created with the same digest), or, alternatively, the new leader inserts a special PRE-PREPARE messages with a `null` digest for sequence numbers where this is not the case.

Upon receiving a $\langle$NEW-VIEW$, v + 1, seq, V, O\rangle$ message, each replica verifies its correctness by checking if it is properly signed, if the VIEW-CHANGE messages in V are valid, and compute O in the way the leader did. After successful validation, each replica appends this information to its log, broadcasts a PREPARE message for every PRE-REPARE in O, and enters view $v + 1$. At this point, the protocol continues in the normal case operation with replicas repeating the protocol steps for messages between the min and max sequence number included in O, but avoid execution of client requests that have already been executed before. If a replica misses a stable checkpoint or some client request, it can obtain missing information from some replica that certified the correctness of it in V (at least $t + 1$ of these must be correct).

PBFT Guarantees SMR Safety

In PBFT, replicas employ a two-phase commit: In both the PREPARE and COMMIT stage each replica forms a quorum of $2t + 1$ replicas (including itself) to make the decision to advance (Castro and Liskov 1999):

Prepare. The PREPARE stage is sufficient to ensure safety for correct replicas in the *same view.* This is because if a single correct replica r_i *prepared* s for m in view v, then s can not be prepared for any other $m' \neq m$ in v. The reason for this is that the *prepared* predicate implies r_i gathered a quorum Q_i of PREPARE votes. Suppose another correct replica r_j prepares s for $m' \neq m$ in v. Then r_j employs some quorum Q_j. Because both quorums are sufficiently large to intersect in at least one correct replica (since $|Q_i \cap Q_j| \geq t + 1$), it means a correct replica has equivocated a PREPARE vote. This is a contradiction (correct replicas never equivocate), so it follows that $m = m'$.

Commit. The COMMIT after PREPARE ensures safety for correct replicas across *changing views.* The reason for this is rooted in the mechanics of a two-phase commit. If a single correct replica r_i *committed* s for m in v, then it implies r_i gathered a quorum Q_i of COMMIT votes. This means a majority of $2t+1$ replicas (in particular among them $t+1$ correct replicas) have already *prepared* s for m in v. Since s is prepared for m by the majority of correct replicas $(t+1)$ in the system, these will later refuse to prepare any other $m' \neq m$ for s *even in higher views*, so there can never be a quorum Q_j formed to support m' (in the same view, trivially, since that would imply there can be only $n - |Q_j| = t$ votes for m but we know already there must be $t+1$ votes for m from the correct replicas in Q_i). Now, the view-change protocol plays an important role: Replicas do not process a PRE-PREPARE message for any higher view $v' > v$ before having received a NEW-VIEW message first. Since any acceptable NEW-VIEW message would contain VIEW-CHANGE messages from a quorum Q' of replicas, both quorums would intersect in at least once correct replica (because $|Q_i \cap Q'| \geq t+1$). That means, there must be one correct replica that both (1) prepared s for m in v and (2) included this fact (i.e., supporting s for m) in its VIEW-CHANGE message contained in NEW-VIEW. So, the NEW-VIEW message ensures that this information verifiably propagates to all higher views $v' > v$. Now, a new leader for $v' > v$ may re-run the agreement protocol and is then forced to assign the same sequence s for m. Note, that any other $m' \neq m$ for s will be rejected by all correct replicas in v'. Alternatively, the NEW-VIEW message may contain a valid checkpoint for some sequence $s' > s$. Then, for sequence numbers lower than s' no message will be committed anymore. That is because the execution of these requests is then already reflected by the checkpoint state (and $\geq t + 1$ correct replicas already executed m's operation at position s).

PBFT Guarantees SMR Liveness

To guarantee liveness, PBFT additionally assumes partial synchrony (i.e. some unknown upper bound on message processing and delivery). In particular, PBFT's liveness relies on sufficiently enough ($2t + 1$) correct replicas to *eventually enter the same view*. While PBFT does not come with a formal proof for liveness, it sketches these three techniques it employs (Castro and Liskov 1999):

Increasing timeouts: A replica starts a timer θ after receiving $2t + 1$ VIEW-CHANGE messages. If θ expires before receiving a valid NEW-VIEW message, then θ is doubled for the next view. Increasing the timeout (exponentially) allows the protocol to (swiftly) adapt to the unknown upper time bound that is assumed to eventually hold in the system. This mechanism maximizes the available time window for replicas to arrive in the same view.

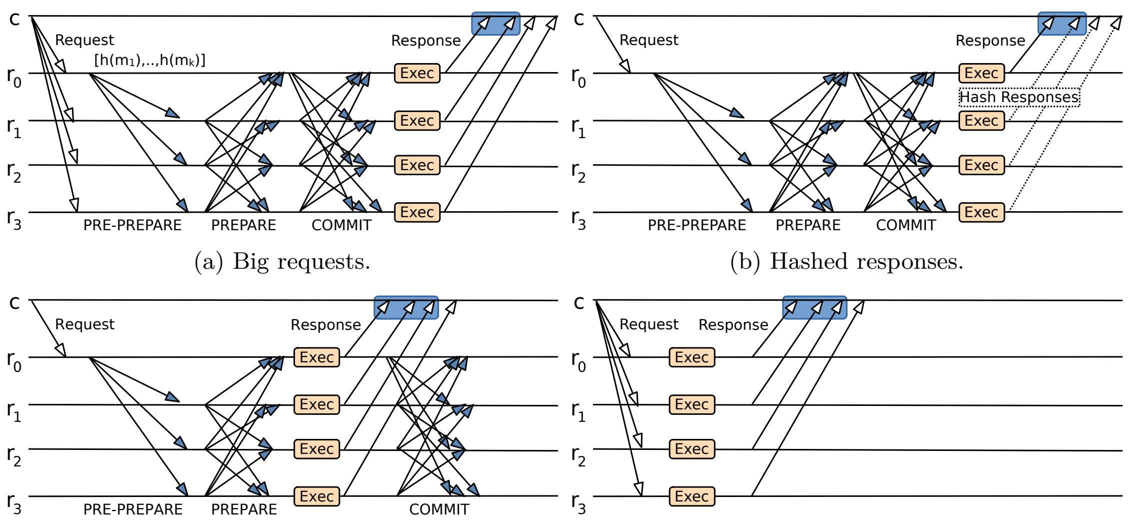
Catching up with others: Once $t + 1$ VIEW-CHANGE messages are received from others for *higher views* than its current, a replica broadcasts a VIEW-CHANGE for the smallest view in this set. This is because it is then safe to assume that the replica is lacking behind some (at least one) correct replica. Picking the smallest view number is a safe decision to avoid possible tampering from Byzantine replicas.

Preventing perpetual view changes: Because of the requirement of $t + 1$ VIEW-CHANGE messages to trigger a view-change, the t faulty replicas can not purposely trigger it in the non-leader role. Even if we assume cascading leader failures, then after t view changes, the system will finally end up in a view with a correct leader because of the round-robin leader selection which first probes through every replica as possible leader before selecting some specific replica for a second regency.

Optimizations

To achieve performance similar to a non-replicated system, PBFT comes with a set of performance optimizations, for which we display the behavior of PBFT under these optimizations in Figure 2.8 (Castro and Liskov 1999; Castro 2000):

- *Request batching*: The costs of running the three-step agreement protocol can be amortized over multiple requests. The leader batches multiple requests into a single batch and then includes it in the PRE-PREPARE message instead of just a single client request.
- *Message authentication codes*: PBFT is originally described using signatures, but the implementation employs message authentication codes which are faster to compute and verify.



(a) Big requests. (b) Hashed responses.

(c) Tentative executions. (d) Read-only requests.

Figure 2.8: Optimizations introduced in PBFT (Castro and Liskov 1999).

- *Big request transmission*: The client broadcasts big requests to all replicas (instead to just the leader). The leader only includes the hash of these requests in the PRE-PREPARE message.
- *Hashed responses*: All but one randomly chosen replica (designated by the client) respond with the hash of the result. The client can collect the hashes to verify the correctness of the response. If the chosen replica is faulty, then the client re-submits its request after some timeout and demands all replicas to include their result.
- *Tentative executions*: Replicas *tentatively* execute requests after the *prepare* phase and directly respond to the client. The application state needs to be reverted if a leader failure happens. Further, the client needs to gather $2t + 1$ replies to a tentative request.
- *Read-only requests*: Read-only requests bypass the agreement protocol and are directly served by the replicas. If a client fails to gather sufficient matching replies, it resubmits the request as a standard read-write request. This requires a client to wait for $2t + 1$ responses (for read *and* read-write operations) instead of $t + 1$ to preserve linearizability under this optimization.

PBFT has been a ground-breaking work in the field of practical BFT protocols and was followed by a lineage of approaches that aimed to further improve it (Cowling et al. 2006; Wester et al. 2009; Clement, Kapritsos, et al. 2009; Clement, Wong, et al. 2009; Guerraoui, Knežević, et al. 2010; Amir, Coan, et al. 2010; Aublin, Mokhtar, et al. 2013; Kapitza et al. 2012; Bessani, Sousa, et al. 2014).

In the next subsection, we review a more recent BFT SMR library, BFT-SMaRt (Bessani, Sousa, et al. 2014) in more detail, because it serves as a research prototype used for contributions this thesis makes.

2.3.3 BFT-SMaRt

A still maintained BFT library is BFT-SMaRt (Bessani, Sousa, et al. 2014), which is a Java-based open-source library that offers durable and customizable BFT SMR. Compared to other SMR implementations such as UpRight (Clement, Kapritsos, et al. 2009) or PBFT, BFT-SMaRt offers a number of advantages including a dynamically scalable replica set as well as a modular architecture with separate components for different functions like state transfer, reconfiguration, and consensus.

Another advantage is the high performance it achieves through its multi-core design and optimization techniques. Additionally, BFT-SMaRt is much more configurable than its competitors. For example, it can run in either crash fault tolerance (CFT) mode or BFT mode, with fewer replicas and a faster operation in CFT mode (2 protocol steps for consensus instead of 3).

Mod-SMaRt assumes a consensus protocol abstraction called *Validated and Provable Consensus (VP-Consensus)*. In addition to normal consensus protocols, it has an interface for handling timeouts which can be triggered by the SMR protocol and has one additional property called *external provability* and a modified validity property (Sousa and Bessani 2012):

Definition 2.3.4 **VP-Consensus**

VP-Consensus *satisfies the properties of consensus, and additionally (Sousa and Bessani 2012):*

- *External Validity: If a correct process decides v, then a validated predicate $\gamma(v)$ is true;*
- *External Provability: If some correct process decides v with proof Γ in a consensus instance i, all correct processes can verify that v is the decision of i using Γ.*

Normal Case

We explained the programming model of BFT-SMaRt in Section 2.3.1. BFT-SMaRt uses the Mod-SMaRt algorithm (Sousa and Bessani 2012) for implementing SMR based on an exchangeable, leader-driven consensus core (for which a variant of Byzantine Paxos is employed (Cachin 2009)). In particular, the implementation of Mod-SMaRt presents a latency-optimal construction of SMR based on this exchangeable consensus primitive and makes the normal case operation of the overall protocol a three-step pattern that resembles PBFT (although the steps are called PROPOSE, WRITE and ACCEPT, thus similar to Figure 2.6 but with a client sending the request to all replicas).

Modularity is not the only aspect that improves the comprehensibility of Mod-SMaRt over PBFT. Mod-SMaRt also does not parallelize its normal case pattern. Instead, the leader only starts a new consensus instance after the last one successfully completed, and correct replicas do not participate in consensus instances other than their currently ongoing consensus instance which directly succeeds (i.e., is for the incremented id of) the last decision that was made.

Consensus. In BFT-SMaRt[8], the client sends its request to all replicas. Then, in the PROPOSE step, the leader broadcasts a message containing a batch of requests to be decided to all other replicas. The subsequent two communication steps, WRITE and ACCEPT, are all-to-all broadcasts utilized for two-phase commitment. During these steps, each replica i waits for a quorum Q_i made up of $\lceil \frac{n+t+1}{2} \rceil$ replicas to move forward (*a Byzantine dissemination quorum*, see Section 2.1.3). Two different replicas $i \neq j$ may form different quorums, but these quorums must have at least one correct replica in common, i.e., $|Q_i \cap Q_j| \geq t + 1$. The client accepts a result of the operation once it gathered sufficiently many responses (the concrete number depends on the fault model and optimizations used).

The normal case operation of BFT-SMaRt will be successful as long as the leader is correct and no more than t replicas are faulty (and assuming sufficient synchrony). If the leader becomes faulty, the system can still recover its liveness through invoking the *synchronization phase* of the protocol (which fulfills the same purpose as a *view-change* in PBFT but is differently implemented).

Synchronization Phase

In Mod-SMaRt, the synchronization phase replaces the leader, makes all correct replicas synchronize their state and continue in the same consensus instance (Sousa and Bessani 2012) (see also Figure 2.9):

[8]For the sake of improved exposition, we only explain the configuration to tolerate Byzantine faults.

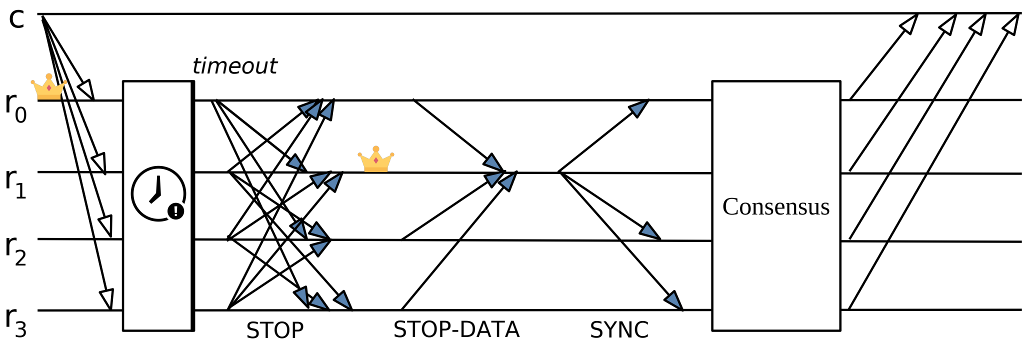


Figure 2.9: *Synchronization phase* of Mod-SMaRt (Sousa and Bessani 2012).

Stopping execution. A replica starts a timer for a client request once it was received. When the timer on some request m expires, a replica forwards m (or a set of requests M) to all replicas and re-starts the timer(s). Once some timer expires a second time, a replica suspects that the current leader must be faulty, stops the execution of consensus instances, and broadcasts a $\langle \textsc{stop}, reg, M \rangle$ message to all other replicas, where reg is a number indicating the next regency and M a set with all timed-out requests. Upon receiving $t+1$ such STOP messages from different replicas (potentially including its own), a replica stops executing consensus instances and broadcasts a STOP message itself, unless it has already done that before. Eventually, all $n-t$ correct replicas will stop consensus execution and know which requests are timed out in the other replicas. Now, existing timers on requests are canceled. Note, that replicas may enter the synchronization phase even under a correct leader if there is a temporary phase of asynchrony in the system.

Installing the new leader. Upon receiving $2t+1$ STOP messages, a replica installs the next leader, which is chosen in a round robin fashion similar to PBFT. Now, the timers on all requests which are still waiting for being ordered are re-started.

State synchronization. Subsequently, a replica sends a $\langle \textsc{stop-data}, creg, Log \rangle$ message to the new leader of the current regency $creg$, in which it includes a Log that contains all decisions it has made so far along with a proof Γ for each decision. The proof can be, for instance, a quorum of signed ACCEPT messages from the consensus primitive. After that, the leader can use the proofs to validate the correctness of decisions. Moreover, the leader collects $n-t$ such STOP-DATA messages and constructs a $Proofs$ object that contains all information collected through the Logs of the others. Now, the leader broadcasts a $\langle \textsc{sync}, creg, Proofs \rangle$ message to all others, which each replica uses to synchronize its decision log up to the highest decided value and resumes decision processing. Finally, replicas execute operations contained in the decisions to reach the same state.

If the synchronization phase completes successfully, replicas can continue running new consensus instances. Otherwise, the synchronization phase is repeated. This can happen if the next leader is also faulty or the network is in an asynchronous phase (e.g., GST has not been reached yet).

2.3.4 Utilisation in Distributed Ledger Technology (DLT)

In this subsection, we define our notion of blockchain and explain the use of BFT as a consensus substrate for DLT on the example of Hyperledger Fabric.

Blockchain

In academic literature, the term blockchain is not uniformly applied. Our definition stresses the reference to an entire system, rather than just a single data structure:

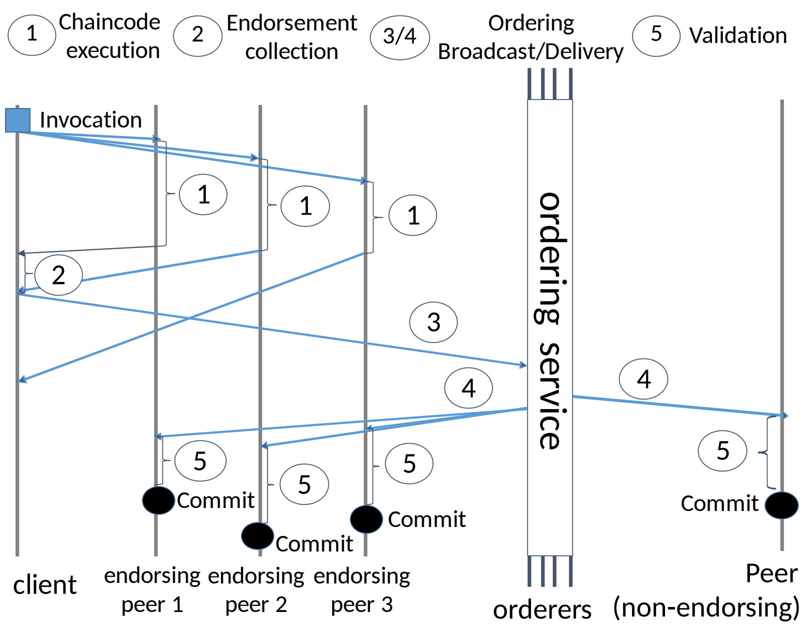


Figure 2.10: Hyperledger Fabric transaction flow (Androulaki et al. 2018).

Definition 2.3.5 **Blockchain**

A blockchain is a replicated, distributed system that maintains an immutable and ordered log of transactions known as the ledger. Multiple replicas hold a synchronized copy of the ledger and participate in validating transactions (in BFT SMR typically called requests or operations) initiated by clients. Typically, a consensus protocol is used to order blocks of transactions. Blocks are linked by including the preceding block's hash in the block header.

The transparent and unchanging history of transactions in a blockchain significantly enhances the reliability and transparency of the system. As long as a substantial portion of the replicas in the system, defined by factors such as number, resource allocation, or stake, act correctly, then the system operates as intended.

In blockchain applications, there is no requirement to rely on the correctness of a single node, thus eliminating the possibility of a single point of failure. Immutability refers to the characteristic where each block contains the hash of the preceding block. If a block is altered, its hash changes, causing the link to break and rendering subsequent blocks invalid (Nakamoto 2008).

Hyperledger Fabric

Hyperledger Fabric (HLF) is an open-source, modular, and extensible blockchain platform designed for permissioned blockchains, in which a consortium of participants is known (Androulaki et al. 2018). Group membership is flexible and managed by a membership service provider, which maps node identities to their public keys. HLF offers a separation of concerns through distinct building blocks and innovates through a *execute-order-validate* architecture, for which traditional BFT SMR protocols can be used in the *order* step. The core idea is that transactions are first simulated (this means executed against the current state of the ledger) by *endorsing peers*, ordered by the *ordering nodes* of the ordering service, and then committed by *committing peers*.

Transaction flow. In more detail, the *transaction flow* consists of the following steps (also shown in Figure 2.10) (Androulaki et al. 2018): A *client* sends a transaction proposal to the endorsing peers that are specified in the endorsement policy and these simulate the transaction execution (①), thus creating read sets and write sets, and does not alter the ledger. Subsequently, the responses of the endorsing peers (②) are collected. They include read and write sets, and the client checks if the endorsement policy is satisfied. In this case, the client puts them in an *envelope* and submits the envelope to the *ordering service* (③). Then, the ordering service orders all transactions. Finally, transactions are batched into a block once one of the conditions for cutting a block is met and then the ordering service broadcasts the block to the committing peers (④). Committing peers validate or invalidate transactions in the block and eventually the block is appended to the *ledger* (⑤).

Integration of BFT-SMaRt into HLF

Sousa and Bessani designed and built a BFT ordering service for HLF using BFT-SMaRt (Sousa, Bessani, and Vukolić 2018). In Chapter 5, we will reuse the same service to integrate our own protocol, AWARE, which uses interfaces identical to that of BFT-SMaRt. In the following, we briefly summarize how this integration works:

HLF offers a client interface to submit envelopes to an external ordering service called *HLF consenter*. These envelopes are sent to a Java *frontend*, consisting of a client thread pool, receiver thread, and an asynchronous *BFT Proxy*. The BFT Proxy is part of the client-side BFT-SMaRt library and broadcasts the envelopes to the ordering nodes, which then order them as they pass through the *total order multicast layer* of BFT-SMaRt. The ordering nodes extend the *ServiceReplica* class from BFT-SMaRt. They receive a sequence of totally ordered envelopes and group them into blocks using a *blockcutter*. The created and signed blocks are then distributed to all receiving *frontend BFT Proxies* using the *Replier* interface of BFT-SMaRt. Once the BFT Proxy collects enough matching and verified messages for a block, it is passed to HLF to be added to the ledger.

Chapter 3

SotA on Planetary-Scale, Fast, and Adaptive BFT SMR

Related research work studied how to improve certain aspects of BFT SMR designs. In this section, we provide an overview of related work, which covers optimizations addressing the geographical distribution of replicas, adaptiveness, and optimistically using fewer rounds for agreement.

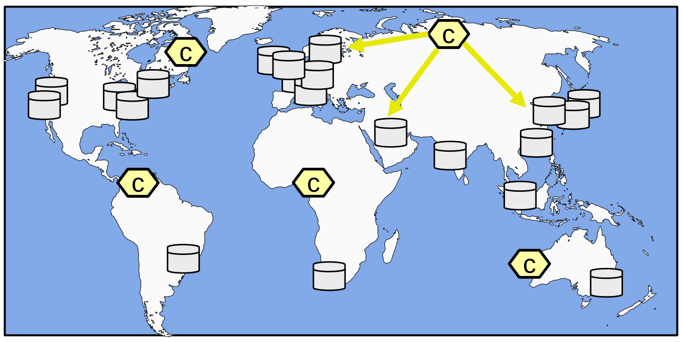


Figure 3.1: A planetary-scale deployment of a BFT SMR system with a geographically dispersed set of clients (c) that *read*/*write* from/to replicas (🛢) distributed across the globe.

3.1 Geographically-distributed SMR

To begin with, a variety of research efforts approached the improvements of SMR for wide-area networks such as displayed by Figure 3.1.

Mencius (Mao, F. P. Junqueira, et al. 2008) presents a methodology to construct highly efficient crash fault-tolerant SMR for WANs by relying on a rotating coordinator scheme. Using this technique, clients submit their requests to a geographically close replica, thus accelerating the client-replica interaction. The skipping technique which it employs can not be easily used when under the assumption of Byzantine faults.

In follow-up work, Mencius was later improved by presenting the RAM protocol (Mao, F. Junqueira, et al. 2009). The RAM protocol inherits the rotating leader design and incorporates attested append-only memory (A2M) in their solution and operates under the assumption of mutually suspicious domains (MSD), in which clients assume their domain (geographically close replicas) to be non-malicious. A2M allows the PBFT-like normal case operation to complete after only one step after witnessing the proposal, while the MSD assumption allows a client to accept a result as final after hearing the response from its local coordinator, thus reducing the latency further.

PBFT-CS (Wester et al. 2009) improves PBFT for WANs by incorporating client-side speculation. In this approach, clients predict a response to a previous request and proceed to send subsequent requests without waiting for the earlier request to be committed. However, clients need to keep track of and communicate the dependencies between their requests (*predicated writes*).

EBAWA (Veronese et al. 2010) is a protocol designed to enhance SMR in WANs by incorporating a hybrid fault model. By using a trusted component on each replica, EBAWA effectively reduces the required number of replicas in the system to $2t + 1$ and minimizes the communication steps needed for agreement to just 2. In addition, EBAWA also employs a rotating leader technique, in which clients submit their requests to geographically close replicas.

Steward (Amir, Danilov, et al. 2010) presents a hierarchical architecture that offers improved latency for SMR protocols in WANs. Its idea is to consider multiple regional replication groups, and each group runs BFT agreement. These groups represent different sites of the system and are finally connected through a crash fault-tolerant protocol. Thus the fault model assumes that Byzantine behavior is bounded within each regional group. Steward's concept of a hierarchical architecture was later employed in the design of the Fireplug framework (Neiheiser, Rech, et al. 2020). Fireplug offers efficient geo-replication of graph databases by composing several BFT SMR groups, implemented with BFT-SMaRt (Bessani, Sousa, et al. 2014).

WHEAT (Sousa and Bessani 2015) can be best described as being an optimized variant of BFT-SMaRt for its use in WANs, inspired by an empirical design. In particular, it improves the latency of BFT SMR through the use of tentative executions and weighted replication. Moreover, the use of weights allows replicas to form proportionally smaller consensus quorums, which can accelerate the time needed for agreement while offering the same resilience as non-weighted BFT. Since this thesis builds largely upon WHEAT as a research foundation, we describe this approach in more detail in Section 3.4.

GeoPaxos (Coelho and Pedone 2018) is an approach for fast, crash fault-tolerant SMR under geographic dispersion. GeoPaxos decouples order from execution, utilizes a partial order instead of total order, and further makes use of geographic locality (where objects are assumed to have a preferred site, at which they are most likely being accessed). Another crash fault-tolerant protocol, Egalitarian Paxos (Moraru et al. 2013), allows all replicas to propose and uses a mechanism to solve conflicts between interfering operations.

Further research work studied the evaluation and ranking of replica deployments in geographic SMR to indicate well-performing system configurations for a given WAN topology to a system administrator before the actual runtime (Numakura et al. 2019).

A recent advancement in crash-fault tolerant SMR optimized for wide-area replication is Weave (Eischer, Straßner, et al. 2020). Weave offers low-latency execution guarantees for client requests by eliminating the need for clients to wait during wide-area communication steps. This can be accomplished by utilizing a partitioning of the system into local groups, where each consists of $t+1$ replicas, which assign sequence numbers to write requests submitted from their local clients.

Moreover, Spider (Eischer and Distler 2020) is another optimization for WANs that assumes a model inspired by the common architecture of public clouds, which distinguishes between regions and availability zones. Spider's core idea is to separate agreement from execution and employ execution groups at each system site (i.e., each region). The agreement protocol is run by a dedicated agreement group which consists of replicas within a single region and thus can benefit from fast intra-region communication links. Finally, groups are connected over a reliable group-to-group message channel to achieve strong consistency in the system.

DispersedLedger (Yang et al. 2022) is an asynchronous BFT protocol that can improve system performance in WANs with variable bandwidth. It operates on the fundamental concept of enabling nodes to propose, order, and reach consensus on transaction blocks without requiring the complete download of their contents. This approach allows nodes to establish agreement on an ordered log of blocks and ensures the availability and immutability of each block within the network.

Orion (Yahyaoui et al. 2024) is a novel hierarchical approach to geo-distributed BFT replication. In Orion, geographically close replication groups form *clusters* which run a BFT protocol (i.e., HotStuff (Yin et al. 2018)) and each cluster has a representative in the global group. Subsequently, a consistent broadcast primitive (Bracha 1987; Bonomi et al. 2021) is used to disseminate blocks to all other clusters, and finally the global group uses the Damysus protocol (Decouchant, Kozhaya, et al. 2022) to decide on a super-block in a global consensus instance.

3.2 Adaptive BFT SMR Designs

Several research papers explored how to make BFT protocols adaptive, for instance, to optimize performance, or to prevent performance degradation - both under changing environmental conditions (i.e., faults happening or changing network characteristics).

Redundant Byzantine fault tolerance (RBFT) (Aublin, Mokhtar, et al. 2013) is an approach in which system performance is being monitored under redundant leader executions (albeit only a single instance is used to determine the ordering of requests). RBFT allows the system to analyze whether the currently configured leader is fast enough in comparison with system configuration in which some other replica acts as leader. In particular, a faulty leader that purposely degrades performance can be thus identified and replaced, thus preventing a performance degradation of the system.

Abortable state machine replication (Abstract) (Aublin, Guerraoui, et al. 2015) presents a design methodology aimed at simplifying the development and reconfiguration of abortable replicated state machines. Unlike traditional replicated state machines, these machines can abort client requests and delegate this responsibility to another Abstract instance. The design approach offers flexibility by allowing the system designer to determine the criteria for optimization, commonly referred to as the “common case”, for instance, involving assumptions such as the absence of faulty replicas or minimal request contention. Abstract allows adaptiveness by optimistically attempting a BFT protocol that is well suited for the common case, and falling back to a more resilient backup protocol and thus ensuring liveness in case some optimistic assumptions of the common case are violated.

ADAPT (Bahsoun et al. 2015) is a system that strives to make BFT protocols adaptive by combining abortable state machine replication with a novel BFT evaluation process that assesses how well profiles of BFT protocols match certain given conditions using mathematical formulas and machine learning. This allows ADAPT to switch between different BFT protocols striving to optimize performance for four impact factors: number of clients, request size, response size, and faulty replicas. In comparison with RBFT, ADAPT does not monitor the current performance of the running, currently executing, protocol.

Adaptiveness can also be utilized when trying to reduce the latency in geographic, crash fault-tolerant SMR under imbalanced localized workloads. The protocols Droopy and Dripple (S. Liu and Vukolić 2017) use a dynamic approach, in which the selection and number of leaders are adapted based on the replica configuration and workload, and thus can vary over time. Droopy dynamically adjusts the leader set for each partition, while Dripple handles the coordination of state partitions.

Moreover, another research work studied the dynamic adaptation of Byzantine consensus protocols (Carvalho et al. 2018) and in particular explores how to port dynamic adaptation techniques that were originally proposed for the crash fault model to the Byzantine fault model. The work further presents an evaluation and comparison of the performance of different protocol switching algorithms (implemented in the BFT-SMaRt library) in practical LAN and WAN setups.

ARCHER (Eischer and Distler 2018) is a protocol crafted specifically for latency-aware leader selection in geo-replicated systems. This protocol leverages the observed end-to-end response latencies from clients to identify and select the optimal leader. By utilizing this approach, ARCHER can dynamically adapt to varying workloads, ensuring efficient performance in the system.

Further research work studies how state transfer can be improved for geo-replicated state machines by adapting to network bandwidth (Chiba et al. 2022). In this approach, the state is divided into chunks and assigned to replicas depending on their dynamically available bandwidth. The work also proposes the dynamic replacement of replicas to reduce latency degradation when network problems occur.

Apart from performance improvements, adaptiveness can also be used to strengthen system *resilience* by reacting to perceived threat-level changes as done by ThreatAdaptive (D. S. Silva et al. 2021). ThreatAdaptive is a BFT SMR system that assumes an available threat detector and adapts to an increase in adversarial strength (i.e. compromised replicas) by increasing the fault threshold parameter t (i.e., by joining additional replicas, thus increasing n and t). When the perceived threat level is low the ThreatAdaptive system can stay in (or return to) a more optimized configuration since a smaller system size usually benefits performance. Further related research shows how to make SMR network-agnostic (Blum et al. 2020), i.e., tolerating a higher threshold of faulty replicas in purely synchronous networks.

Moreover, another research work (Nischwitz et al. 2022) extends one of the contributions made in this thesis, AWARE, applying its prediction scheme with minor modifications to the HotStuff protocol (Yin et al. 2019) to showcase possible performance gains under harsh network conditions (in particular the impact of increased and varying network delays).

Fluidity (Köstler et al. 2023) can relocate replicas in geographic SMR systems to minimize the latencies that clients observe. The Fluidity system adapts to changing workloads as it considers the clients' workload to vary over the period of a day, monitors network characteristics between clients and replicas, and attempts to locate replicas close to regions from which most of the workload originates, thus reducing request latency.

3.3 Fast BFT SMR Variants

Employing additional redundancy by utilizing a less-than-optimal fault-tolerance threshold allows developing *fast* (meaning the number of communication steps is reduced) BFT consensus and also fast BFT SMR variants.

For example, two-step Byzantine consensus, e.g., FaB (Martin and Alvisi 2006) and related approaches revisiting FaB (Kuznetsov et al. 2021; Howard et al. 2021) and one-step asynchronous Byzantine consensus in contention-free scenarios (Friedman et al. 2005; Song and Renesse 2008) have been proposed and shown to be feasible.

In general, the term "fast" needs to be taken with a grain of salt, because due to the added redundancy and proportionally larger quorums (which require collecting votes from all over the world), such fast variants of protocols may be slower than their "classic" counterparts in some real-world scenarios as it has been shown in an analytical comparison for Paxos vs. FastPaxos (F. P. Junqueira, Mao, et al. 2007).

Besides, BFT SMR protocols like Quorum and HQ replication have been developed that can, in contention-free scenarios, operate through only client-sever interaction and optimistic execution (Abd-El-Malek et al. 2005), or suggest a hybrid approach in which — when under contention — employ PBFT's agreement pattern (Cowling et al. 2006).

PBFT's implementation of read-only requests suggests an optimistic two-step protocol execution for operations that do not alter the state. Further, PBFT's proposed *tentative execution* optimization can tentatively reduce the number of steps in the protocol by one (see Section 2.3.2 for details) (Castro and Liskov 1999). Another optimization, *speculative execution* allows replicas to execute and respond to client requests after receiving the proposal as introduced in Zyzzyva (Kotla et al. 2007). These optimizations are not easy to implement, as it has been shown that there is a safety violation in Zyzzyva and a liveness violation in FaB (Abraham, Gueta, et al. 2017) and a liveness violation when using the read-only optimization of PBFT (see Chapter 4).

3.4 WHEAT: An Empirical BFT SMR Design for Planetary-Scale Systems

WHEAT is an acronym for WHeight-Enabled Active replicaTion and presents an empirical approach developed to enhance the performance of BFT SMR for deployments in which replicas are spread across diverse geographical locations (Sousa and Bessani 2015).

A fundamental finding of WHEAT is that by incorporating Δ additional replicas, which do not impact the resilience threshold (i.e., $n = 3t + 1 + \Delta$), client latency can be effectively minimized.

But why is that the case?

The reason behind this is that quorums in WHEAT are not formed as the usual *threshold*-based Byzantine dissemination quorums (see Section 2.1.3) which are employed in related research work on BFT SMR (Castro and Liskov 1999; Clement, Kapritsos, et al. 2009; Veronese et al. 2010; Bessani, Sousa, et al. 2014; Yin et al. 2019). Instead, quorums are formed using weighted replication, which allows to form quorums of variable size (see Section 2.1.3, Figure 2.2b). In particular, proportionally smaller quorums can emerge in a weighted quorum system.

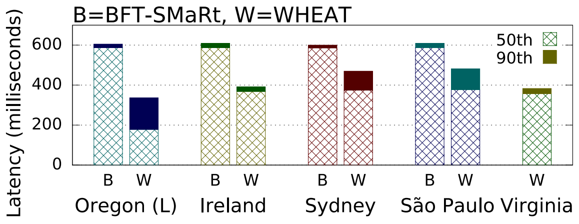


Figure 3.2: Latency improvement using weighted replication in WHEAT vs. BFT-SMaRt, as observed by clients co-located with different replicas (Sousa and Bessani 2015).

Example. This can be showcased in an example: Since BFT systems typically form Byzantine t-dissemination quorums of size $\lceil \frac{n+t+1}{2} \rceil$ replicas as displayed in Figure 2.2a, with $n = 5$ replicas (and assuming $t = 1$), it means the quorums size needs to be 4 to guarantee intersection in at least $t + 1$ replicas.

However, it is possible to satisfy the *same intersection property* by designing the quorum system to favorably intersect in specific replicas, which is realized by assigning higher *weights* to these specific replicas as we can see in Figure 2.2b. Rather than a concrete size of replicas, these quorums need to be comprised of replicas whose accumulated weights equal a concrete total sum of weights.

In our example from above, a fast quorum can contain only 3 replicas (see the blue quorum in Figure 2.2b, which includes replicas with a total voting weight of 5). Thus, the size of this quorum is less than that of an egalitarian quorum, but it ensures intersection by including all replicas with high voting power.

Let's consider a scenario where the fastest and geographically closest replicas are part of this specific quorum. These replicas have the advantage of advancing the voting stages of consensus more quickly since they have to wait for a shorter time to gather the required voting weights. By making use of additional spare replicas and speeding up consensus, WHEAT can effectively reduce the system's overall latency as can be seen in Figure 3.2.

WHEAT quorum system. As mentioned before, WHEAT does not wait for a predefined number of replicas but waits for a predefined sum of votes of $Q_v = 2t \cdot v_{max} + 1$ (Sousa and Bessani 2015).

To distribute these weights (or voting power) among the replicas, WHEAT employs a bimodal scheme that distributes the voting weights so that the $2t$ best-connected replicas possess a voting weight of v_{max} which is calculated as:

$$v_{max} = 1 + \frac{\Delta}{t}$$

All remaining replicas possess a voting weight of 1. Let $w(s)$ denote the weight of a replica s. The construction of a dissemination quorum system is given by

$$\mathcal{Q} = \{Q \in 2^U : \ |Q| \leq n - t \wedge \sum_{s \in Q} w(s) \geq 2tv_{max} + 1\}$$

More details on the calculation and correctness of this quorum system can be found in the WHEAT paper (Sousa and Bessani 2015). Note that the concrete size of every possible quorum in this system varies between $2t+1$ (minimum to satisfy intersection) and $n-t$ replicas (necessary for availability when considering the system to be fail-prone). It is important to notice that the size of the smallest possible quorum now only depends on the resilience threshold t, but not on the system size n.

By employing this quorum scheme, a WHEAT system guarantees that even in the event of failure of t best-connected replicas, an available fallback quorum exists, which comprises all of the remaining replicas.

Empiric design. Apart from using this weighting scheme, WHEAT also uses an optimization that has been chosen from a set of possible optimizations after an empirical evaluation of different optimizations in a real-world evaluation and analyzing the trade-offs of incorporating these optimizations (Sousa and Bessani 2015).

This optimization is called *tentative executions* and it was first introduced by PBFT (Castro and Liskov 1999) (see Section 2.3.2 and Figure 2.8c for details). The optimization allows replicas to tentatively execute requests after completing the WRITE stage (or, PREPARE stage in PBFT parlance). WHEAT further employs the read-only optimization that already exists in BFT-SMaRt.

Summarizing the insights from above, the WHEAT approach brings advantages in a geographically dispersed system, because it allows clients to receive replies sooner, thus mitigating the problem of high request latencies from which BFT SMR systems usually suffer.

3.5 Open Problems

In the preceding sections, we reviewed a substantial body of research that focused on enhancing geographically distributed BFT SMR systems and developing adaptive designs for these systems. Despite these advancements, we believe several significant challenges remain unresolved. Some of these challenges have recently emerged with the rise of blockchain systems, a highly relevant application of BFT SMR, while others persist from earlier research, such as issues related to WHEAT (which still serves as a state-of-the-art prototype to explore the effect of optimization techniques on WAN performance). This thesis addresses the open problems discussed below, with a specific focus on optimizing BFT SMR for planetary-scale systems. Our high-level objectives for BFT SMR research include improving latency and making systems adaptive to dynamic conditions.

No automatic selection of system configuration or optimization. To begin with, in many BFT SMR protocols, there is no automatic selection of the best system configuration. This means that the system does not autonomously determine or choose the optimal configuration for its operation. To name a few examples for a potential optimization in WANs, this could include leader selection in protocols like PBFT, BFT-SMaRt and HotStuff, leader selection *and* weight distribution in WHEAT, or the concrete structure of a communication tree in Kauri (Neiheiser, Matos, et al. 2021). Instead, in most BFT SMR frameworks, the configuration must be specified manually by human operators, i.e., a system administrator, who might be overwhelmed with the task of tuning the performance of a notoriously complex BFT system. This is because assessing the performance of a replicated system requires in-depth knowledge about the replication algorithm as well as the network characteristics, in particular, the latencies of communication links.

The lack of automatic selection implies that decisions regarding the system's configuration, such as the selection of replica weights and leader placement, require human intervention and are not dynamically adjusted or optimized by the system itself. However, some events, in particular perceived network changes or faults, might make adaption during runtime a necessity to maintain high system performance.

Larger systems make decisions more difficult. Even if we assume that certain events or failures would not occur in a "perfect world system", the procedure of manually selecting a fast configuration becomes challenging in systems with a large number of replicas. This challenge arises because tuning the configuration of a BFT system typically involves a significant number of possibilities available[1]. Consequently, choosing an optimal system configuration becomes a non-trivial task.

It is noteworthy that novel BFT protocols are largely crafted with the ambition to be implemented within blockchain systems, and thus the *scalability* of an optimization procedure, i.e., a procedure that allows all replicas to autonomously self-optimize becomes a relevant challenge.

Fast reads break SMR liveness. Fast read operations are realized as an optimistic two-step pattern, in which replicas directly respond to clients (see Figure 2.8d). Further, fast read operations can significantly improve latency in geographic BFT SMR, in particular, when an application uses reads more often than writes.

However, the read-only request optimization, which was introduced in PBFT over 20 years ago, can compromise the liveness of the protocol, i.e., the optimization can lead to situations where the system becomes unresponsive.

It is important to note that this issue not only affects the optimized read-only operations but also the standard, totally-ordered operations in PBFT. The vulnerability arises from the fact that a malicious leader can block legitimate clients from making progress within the protocol as the client response quorum needs to be increased to $\lceil \frac{n+t+1}{2} \rceil$ instead of only $t+1$ to satisfy linearizability. By exploiting this weakness, a malicious leader can block correct clients from having their operations successfully executed, thus disrupting the system's liveness.

Since PBFT served as a blueprint for BFT SMR design, this conceptual flaw in the design of optimized-PBFT was inherited in many BFT SMR frameworks that followed.

Weighted replication trades resilience for performance. In WHEAT, one of the requirements for achieving faster performance through weighted replication is the provision of additional spare replicas. Let us consider a system with a fixed number of replicas and available regions. In this system, it is possible to utilize some replicas as spare replicas to enhance system speed. However, doing so would entail compromising optimal resilience. The system administrator faces a trade-off situation where they must decide the level of provided redundancy. This decision can be risky, as it needs to strike a balance between performance and resilience.

A significant aspect of this problem is that the decision is made before the system's runtime and has a direct impact on the quorum system and consequently the system's resilience. Importantly, the system lacks awareness of the "threat level" and cannot effectively counter safety violations caused by malicious replicas if the required level of resilience was initially underestimated. This

[1]Often the number of configurations that could be probed is combinatorial in the number of replicas, an example is WHEAT were a subset of replicas needs to be chosen to be assigned a high voting weight, or Kauri, where a subset of replicas are nominated as internal tree nodes.

illustrates the challenge of making decisions in advance without real-time knowledge of potential threats or attacks.

Missing integration with other optimizations. One aspect that can further enhance the performance of highly optimized systems like BFT-SMaRt and WHEAT is the integration of other optimization techniques. These additional techniques can be explored and tested to improve the speed and efficiency of these systems, even beyond their already optimized state.

For example, in addition to weighted replication, a client could consider utilizing the sum of weights from the responses it has already collected to make speculative decisions and accept results on a speculative level. While this may result in a relaxation of the usual strong consistency guarantee, known as linearizability, it can allow for the completion of less security-sensitive operations with a weaker consistency guarantee.

Another optimization possibility is the incorporation of pipelining consensus instances, which enables the processing of multiple consensus instances in parallel. This means that a leader can propose client requests without waiting for the previous consensus instance to complete, thereby improving the system's overall efficiency.

By integrating these and other orthogonal optimization techniques into the existing feature-rich architecture of systems like BFT-SMaRt and WHEAT, it is possible to explore further performance improvements and explore new trade-offs between consistency, latency, and throughput.

Exploring realistic deployment scenarios. To showcase the advantages of the optimized system in various realistic deployment scenarios, such as geo-distributed deployments across multiple AWS regions, it is crucial to conduct thorough evaluations.

Given that BFT SMR is increasingly employed as a consensus substrate in blockchain systems, it becomes essential to consider larger networks and geographically-dispersed environments. This is particularly relevant due to the unique challenges and requirements associated with such systems. Mostly, related research work on geo-distributed SMR (see Section 3.1) does not conduct experimentation in large systems and with many geographic regions.

To accomplish this, leveraging state-of-the-art simulation and emulation tools becomes highly valuable. These tools enable researchers and developers to explore multiple realistic deployment scenarios and validate the performance of the optimized system in diverse environments. By utilizing these tools, one can gain valuable insights into how well the optimized system performs and behaves under various conditions, providing confidence in its effectiveness and suitability for real-world deployment.

Chapter 4 Fast, Live and Linearizable Reads in BFT SMR

PBFT is a state machine replication protocol that is widely recognized for its high performance in realistic environments. It allows a group of replicas to emulate the behavior of a single centralized server thus making the service mask failures of (a threshold of) individual replicas.

One of the reasons PBFT achieves such high performance is due to a set of optimizations introduced in the protocol (see Section 2.3.2). One notable optimization is the handling of *read-only* requests. In PBFT, a read-only request is a type of client request that does not modify the state of the replicated state machine but only retrieves information from it. Since the read operation does not alter the state, it can be always safely executed by the replicas. In fact, replicas may execute a read operation while using a different order of requests.

To reduce the latency of read operations, PBFT introduces an optimization that bypasses the three-step agreement protocol, allowing the replicas to execute and respond directly to clients. Instead of five communication steps typically required for general client requests, read-only requests in PBFT *optimistically* only require two communication steps. This reduction in communication steps results in lower latency for read operations, making PBFT's performance more comparable to non-replicated systems in practical scenarios (although clients still need to collect a *response quorum* instead of accepting a result from a single server).

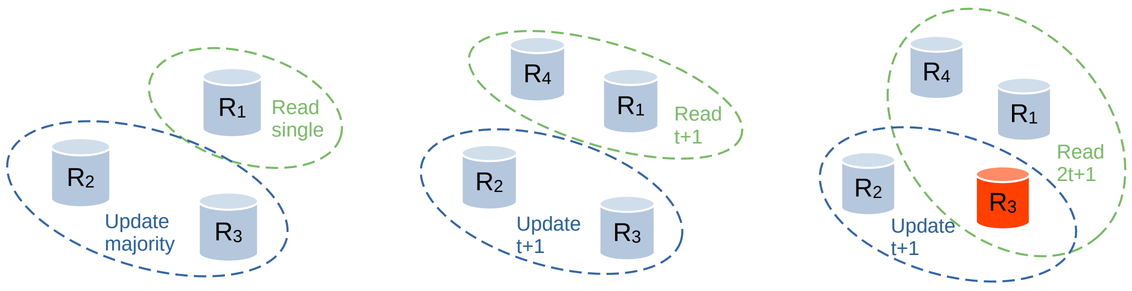
Reading application state through an optimistic read-only request can fail. This is because replicas may execute the read operation in different orders thus producing different results, and, making it impossible for the client to collect the mentioned response quorum. In that case, a client can re-submit the read operation as a standard, totally-ordered operation.

As the design and optimizations introduced in PBFT had a significant impact, its influence can be seen in many subsequent BFT protocols and systems that followed its principles and ideas. One such example is BFT-SMaRt, a BFT protocol that builds upon the design and optimizations of PBFT to provide practical and efficient state machine replication.

This chapter follows for the most part the insights published in an earlier research article:

> Christian Berger, Hans P. Reiser, and Alysson Bessani (2021). "Making Reads in BFT State Machine Replication Fast, Linearizable, and Live". In: *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pp. 1–12. DOI: 10.1109/SRDS53918.2021.00010,

The chapter explains how the read-only request optimization introduced in PBFT more than 20 years ago violates its liveness. The issue is significant because it impacts not only the optimized read-only operations but also the standard, totally-ordered operations within the system when trying to maintain linearizability. To demonstrate this vulnerability, we use this chapter to first present an attack in which a malicious leader blocks correct clients. Additionally, we later propose solutions to correctly integrate the two-step read-only operations in BFT SMR systems on the



***(a) Read locally** in CFT can lead to stale reads.*

***(b) Read and Update t+1** replicas can lead to stale reads in BFT.*

***(c) Read 2t+1 and Update t+1** replicas can lead to stale reads in BFT if there are failures.*

Figure 4.1: Response quorum intersection of read-only optimizations for SMR.

example of BFT-SMaRt (and also applicable to PBFT), thus preserving the liveness and linearizability of the optimized protocol.

4.1 Optimizing Reads in Replicated Systems

To begin with, we take a step back and study how reads are being handled in replicated systems that assume mixed read/update workloads, thus distinguishing between these two types of operations.

Nowadays, many practical replicated systems tend to execute much more reads than updates, i.e., Zookeeper or Spanner (Hunt et al. 2010; Corbett et al. 2013). At first glance, executing a consensus protocol may appear overly expensive when a client's intention is solely to read a portion of the replicated service's state. This becomes especially evident in wide-area deployments, where the time taken to run a comprehensive five-step protocol like PBFT can be substantial.

Due to this reason, crash fault-tolerant (CFT) SMR-based systems often incorporate approaches to enable reading from a single replica. However, directly reading from a single replica straightforwardly may negatively affect the system's consistency guarantee.

The underlying issue stems from the fact that, in a non-synchronous system model, an operation (such as an update) is considered committed and ready for execution when it is confirmed by a quorum of replicas.

With this requirement, a local read that occurs after an update is executed, as depicted in Figure 4.1 (a), may not accurately reflect the committed update. Rather than that, a client may observe that an old value is read. Note that this is not necessarily a problem in many practical applications that can tolerate reading stale values. In particular, the optimization decipted in Figure 4.1 (a) is employed in Zookeeper (Hunt et al. 2010) and allows a client to read from any replica.

Nevertheless, employing this optimization in conjunction with FIFO communication channels restricts the system's compliance to sequential consistency (Attiya and Welch 1994) rather than linearizability. In other words, clients may read outdated values, but these values are part of a consistent sequential history of operations (F. P. Junqueira, Reed, et al. 2011). By exclusively adhering to sequential consistency, Zookeeper can enhance its scalability and more effectively handle read requests by leveraging the number of replicas it possesses. On top of improved scalability, single-replica reads may be served from a geographically close replica which benefits a client's observed request latency.

To maintain linearizability while still allowing single-replica reads, certain CFT systems use temporary leases. The widely adopted approach involves exclusively reading from the stable Paxos leader (Corbett et al. 2013). However, there are also more sophisticated methods available for implementing leases, enabling different replicas to serve reads for distinct parts of the system's state (Moraru et al. 2014).

In the presence of t Byzantine failures, it becomes evident that reading from a single replica is unfeasible, as the obtained result may not accurately reflect the true state of the system. Consequently, a minimum of $t+1$ replicas needs to be accessed. However, even when reading from this number of replicas, it may still result in a violation of linearizability, as depicted in Figure 4.1 (b).

This is because an update operation could have been completed with the responses from $t+1$ replicas, which have committed, and updated to the newest state, but the read operation is served later from the other replicas that have not yet committed the newest value, thus leading to a stale read.

To our current understanding, the sole approach for carrying out a read operation without executing consensus or compromising linearizability is by employing the read-only optimization initially introduced in PBFT (Castro and Liskov 1999).

In PBFT, the handling of read-only requests involves employing a direct request-reply pattern, where all replicas promptly respond to a read without ordering the operation. However, to ensure linearizability, *both read and update operations* complete only when clients collect matching responses from a quorum[1] of $q = 2t+1$ replicas.

We illustrate the reason for this requirement in Figure 4.1 (c): The intersection between a quorum Q_{read} with $q = 2t+1$ replicas and a quorum Q_{update} with only $t+1$ replicas may consist of only Byzantine replicas that can respond a stale value. Further, this requirement was also explicitly stated by Castro and Liskov (they use f to denote faulty replicas instead of t):

"*The read-only optimization preserves linearizability provided clients obtain* $2f+1$ *matching replies for both read-only and read-write operations.*" (Castro and Liskov 2001).

If a client cannot collect q matching replies for some optimized read operation, then it re-transmits this operation as a normal, ordered operation.

4.2 The Problem in PBFT and BFT-SMaRt

The correctness of the optimization discussed in the previous section relies on an implicit assumption made in the system: *Eventually, all correct replicas (at least* $2t+1$*) will execute all updates submitted to the system and can respond to clients.*

At first hand, this assumption sounds very plausible. The number of correct replicas in the system must be indeed at least $2t+1$ (by assumption), and the assumption of eventual delivery might stem from the *Agreement* property of total order multicast (Hadzilacos and Toueg 1994):

If a correct process delivers m, then all correct processes eventually deliver m.[2]

In this section, we show that there is a subtle issue that affects both PBFT and BFT-SMaRt, and we establish an understanding of this problem by first explaining a possible attack in the system.

4.2.1 The Isolating Leader Attack in PBFT

We sketch the attack during a protocol run in Figure 4.2. As usual, a client submits a normal update operation to the system. Subsequently, the Byzantine leader sends a PRE-PREPARE message to replicas r_1 and r_2 but omits to send this message to another correct replica, r_3. In general, the Byzantine leader could omit to send the PRE-PREPARE to a specific group of t correct replicas.

[1]The quorum size of $2t+1$ assumes a PBFT system with $n = 3t+1$ replicas. For arbitrary n and t the quorum size is $\lceil \frac{n+t+1}{2} \rceil$.

[2]Or, alternatively, due to the consensus' Termination property (Hadzilacos and Toueg 1994): *all correct processes eventually decide.*

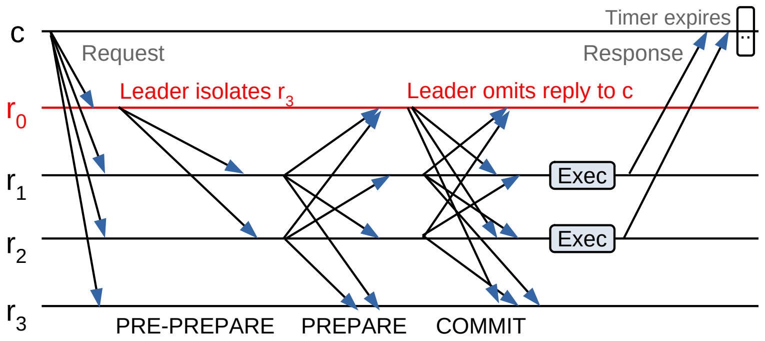


Figure 4.2: In the isolating leader attack, a Byzantine leader separates up to t correct replicas from participation in the normal case operation pattern, thus preventing clients to accept a reply if a response quorum of $2t + 1$ is used.

If we assume a malicious intention, a goal of the Byzantine leader could be to prevent this client from receiving enough replies to accept a response from the system, which could be a kind of censorship attack.

As seen from the perspective of the other replicas (r_1 and r_2), the Byzantine leader follows the normal protocol description. Since the PREPARE messages do not include the proposed value v, but only its digest, the isolated correct replica r_3 does not know the actual proposal as its transmission was omitted before.

In general, the group of isolated replicas (here: r_3) can not prepare v. This means the group of isolated replicas can not proceed to participate in the normal case of PBFT's protocol anymore.

Moreover, when the COMMIT phase is completed through the participation of replicas r_0, r_1, and r_2, the request (or batch of requests) is executed by all replicas that decided on it. We know that in total there will be $2t + 1$ replicas, but among them can be up to t Byzantine replicas as well.

Now, this circumstance can prove to be a striking factor: Imagine that the Byzantine leader (and possibly all other Byzantine replicas) omit to respond to the client. In this case, a client only receives $t + 1$ responses (in our example: from r_1 and r_2).

As we explained before, the read-only optimization requires a client to collect a quorum of $2t + 1$ matching replies to satisfy linearizability. In our example, the client is unable to accept the result and will timeout. Looking at the normal case pattern of PBFT we can now start to assume that the protocol might be stuck, but it is also necessary to regard the other subprotocols of PBFT to fully uncover this liveness issue.

Re-transmission attempts are useless. After a timeout, a client may try to re-submit its operation to the system. In this case, all correct replicas that executed the request (between $t + 1$ and $2t$) re-transmit their result, while the Byzantine leader and other faulty replicas continue to selectively not respond to the client.

In this scenario, all the correct replicas that processed the request (here: r_1 and r_2) retransmit their results, while the Byzantine leader and other faulty replicas persist in selectively omitting responses to the client. It is important to note those correct replicas that did not execute the request can not respond to the client as well.

A view-change will never happen. To address issues arising from a faulty leader in PBFT, a *view change* mechanism is employed, which involves replacing the leader and synchronizing replicas. However, in the case where a Byzantine leader exhibits misbehavior exclusively towards a group of isolated yet correct t replicas, the view change protocol cannot be initiated by this group.

The reason for this is that $t+1$ VIEW-CHANGE messages are necessary for replacing a leader. This requirement is deliberately designed to prevent a group of t Byzantine replicas from maliciously destabilizing the system by triggering view changes.

State transfer cannot help. The affected client cannot rely on state transfer to resolve its issues due to several reasons.

Firstly, if a single client initiates an operation, there is no guarantee that subsequent operations by other clients, in sufficient numbers, will occur to trigger replicas to perform a checkpoint and consequently initiate a state transfer.

Secondly, even if the t correct replicas are transferred with an updated state that includes the effect of the client's operation, they may not be capable of responding to the client since they did not execute the operation themselves. Note that in certain PBFT implementations where client replies are not written in the checkpoint or are discarded to conserve memory, there is no guarantee that the complete replica state necessary to respond to pending requests is being transferred, potentially causing those requests to remain incomplete.

Thirdly, state transfer occurs infrequently, such as every few minutes or hours, and expecting a client to wait that long to complete a single request is impractical and it can be expected that a client triggers a timeout for the request before it could complete.

Summary and Concluding Remarks to the Problem

Ultimately, the $t+1$ non-isolated, correct replicas engaged in the protocol execution cannot differentiate this attack from a successful normal operation. The issue also remains unresolved even with re-transmissions as long as the Byzantine leader continues conducting the attack.

Note that the described attack is not possible in the basic non-optimized variant of PBFT in which a client only waits for $t+1$ matching replies to accept the result of execution. In PBFT's proof sketch for liveness, (there's no formal proof of liveness for this protocol) (Castro and Liskov 2001), the assumption that a client only requires $t+1$ replies is employed to explain why operations eventually complete. It is important to note that in PBFT, it is guaranteed that at least $2t+1$ replicas must participate to commit a request. However, there is no guarantee that all of these participating replicas are *correct* replicas responding to a client. The problem arises only when the system raises the size of the reply quorum to incorporate fast read-only operations while maintaining linearizability.

To conclude, PBFT does not ensure that all correct replicas execute all of the submitted operations after successful ordering, only that eventually their state will reflect all these operations. The main reason for this is the use of state transfer to deal with lossy channels and finite memory, which basically imposes the need for such a mechanism for recovering servers.

4.2.2 The Isolating Leader Attack in BFT-SMaRt

BFT-SMaRt uses Mod-SMaRt, a modular SMR protocol that can be instantiated with any consensus protocol, provided that it meets certain constraints (Sousa and Bessani 2012). The existing implementation of BFT-SMaRt (Bessani, Sousa, et al. 2014) uses a variant of Byzantine Paxos (Cachin 2009), which results in the combined protocol resembling PBFT, at least in normal operation cases (see Figure 2.6 but with the client submitting a request to all replicas).

Because BFT-SMaRt implements the read-only optimization in the same way as PBFT does, it makes the system vulnerable to the attack we described before. Mod-SMaRt comes with a liveness proof that uses the following assumption (last paragraph of Theorem 2's proof (Sousa and Bessani 2012)) (parameter f denotes t): "*... the consensus instance will eventually decide ... in at least*

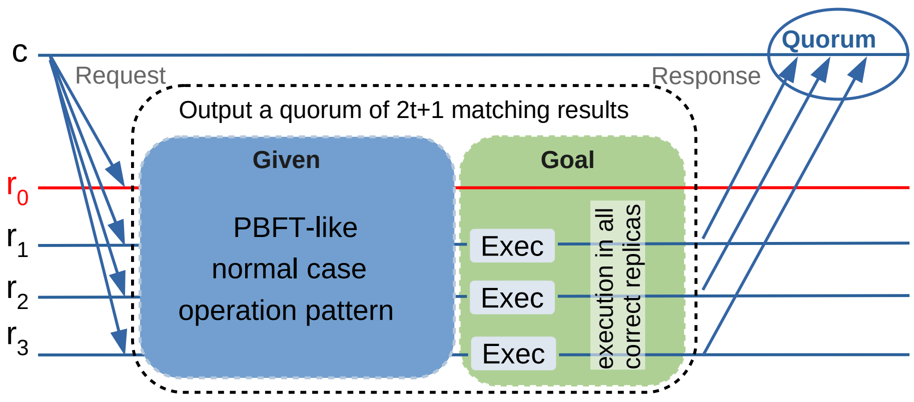

c
Request
Response
Quorum
Output a quorum of 2t+1 matching results
Given
PBFT-like normal case operation pattern
Goal
Exec
Exec
Exec
execution in all correct replicas
r0
r1
r2
r3

Figure 4.3: Abstract view of the problem.

$n - f$ replicas. Because out of this set of replicas, there must be $f + 1$ correct ones, the operation will be correctly ordered and executed in such replicas."

The proof for liveness in Mod-SMaRt shows liveness assuming the system does not use the read-only optimization. But in BFT-SMaRt, the read-only operation is built-in, thus clients always wait for $2t + 1$ replies and are affected by the presented attack as well. Notably, in the modular SMR algorithm, Mod-SMaRt, the embedded consensus protocol itself cannot solve the issue, because Mod-SMaRt only guarantees to start consensus in $n - t$ replicas to begin with.

After our detailed observation of the problem of PBFT's read-only optimization, we see that the main challenge to make the optimized system correct is to make an implicit system assumption valid (see Figure 4.3): A quorum of $q = 2t + 1$ matching replies needs to arrive at the client for any executed operation, which in turn demands the execution of ordered requests in at least q *correct* replicas and that these replicas can respond to pending operations even after a state transfer happened. When there are fewer than q replicas responsive, the use of PBFT's read-only optimization in BFT systems opens a vulnerability in which a Byzantine leader may compromise the system's liveness.

4.3 Linearizable and Live BFT SMR with Fast Reads

In this section, we provide two simple and generic solutions to address the issue. For the purpose of exposition, we explain the protocol modifications using BFT-SMaRt as an example, but the solutions can be also applied to PBFT without many adjustments.

The proposed solutions encompass a series of protocol adjustments that effectively eliminate the possibility of isolating correct replicas during executions of the normal case pattern. In particular, these modifications guarantee that if one correct replica executes a request o, all other correct replicas can either execute o as well or, at the very least provide a response for it. This ensures that the client is assured of receiving a quorum of matching replies, which is crucial for the system's linearizability guarantee when using the read-only optimization.

In the following subsections, we first explain a simple and straightforward solution that satisfies the *atomic broadcast' Agreement* property. The idea is to have each replica broadcast its decision to the others. After that, we present a second, refined solution that allows replicas to query and forward decisions *only if this is necessary*. Having the decision forwarded on demand is desirable because this way we spare overhead in expected fault-free executions. Finally, we elaborate on the implications of checkpoints exchanged during state transfer to guarantee that our protocol adjustments can be implemented with bounded memory.

Further, for a simpler presentation, we assume *reliable channels*, which can be implemented on top of fair-loss channels. If a system without reliable channels is assumed, some adjustments are

Algorithm 2: Decision-Broadcasting at replica r

```
Function decide($c, v$)                                                          1
    if $\neg$ isDecided($c$) then                                                2
        DECIDED$_c$ $\leftarrow$ TRUE                                            3
        retrieve $\langle c, v, \Gamma \rangle$ from LOG                          4
        broadcast $\langle$FWD-DECISION$, c, v, \Gamma \rangle$ to all            5
        deliver($c, v$)  ▷ Execute requests in $v$ when all earlier decisions executed   6
Upon reception of a $\langle$FWD-DECISION$, c, v, \Gamma \rangle$ do              7
    if $\neg$ isDecided($c$) $\wedge$ verify($c, v, \Gamma$) then                 8
        LOG $\leftarrow$ LOG $\cup \{\langle c, v, \Gamma \rangle\}$              9
        decide($c, v$)                                                           10
```

necessary to keep re-transmitting messages.

4.3.1 Broadcasting Decisions

The first simple solution for the problem is the following: To guarantee that a decision propagates during the normal case phase to all correct replicas, we add an additional all-to-all broadcast phase to the normal case phase (see Figure 4.4a, and Algorithm 2).

As soon as a replica decides on a batch of requests (i.e., v) in a consensus instance (or for some sequence number in PBFT parlance) c, then each replica $r \in R$ broadcasts the decision together with a *cryptographic proof* Γ to all others by sending a $\langle$FWD-DECISION$, c, v, \Gamma\rangle$ message. Two additional requirements need to be fulfilled:

1. The proof Γ must be non-repudiable, i.e., verifiable *externally*. This can be implemented by including a set of the size of a quorum that contains valid and signed COMMIT messages in PBFT or valid and signed ACCEPT messages in BFT-SMaRt. Further, a cryptographic function VERIFY(c,v,Γ) accessible by replicas checks if a proof Γ for decision v made for consensus instance number c is valid or not.

2. If some replica r' that has not decided consensus c and receives a $\langle$FWD-DECISION$, c, v, \Gamma\rangle$ message, it checks if the attached proof is valid and if so decides v, executing its requests when all of the requests decided in all previous instances have executed.

Because a correct replica always broadcasts its decision when calling DECIDE() along with the cryptographic proof, the algorithm ensures the propagation of this decision to all correct replicas.

4.3.2 Forwarding Decisions on Demand

Although the *decision broadcasting* solution is effective, it imposes an additional burden on system performance because of its message overhead during fault-free executions. Replicas broadcast their decisions promptly, even if it is unnecessary to do so. To avoid this overhead of broadcasting decisions when the leader is correct, our objective is to transform this scheme into an *on-demand* procedure.

Under this new approach, correct replicas can be queried by isolated replicas to forward a decision to them. To accomplish this, we extend the 3-phase consensus protocol with two supplementary messages: REQ-DECISION and FWD-DECISION. The REQ-DECISION message is employed by potentially stuck replicas to request a decision, along with a cryptographic proof Γ, from other replicas. Conversely, the FWD-DECISION message serves as the corresponding reply, containing the forwarded decision and proof.

Algorithm 3: Decision-Forwarding at replica r

```
Function decide($c, v$)                                                              1
    if $\neg$ isDecided($c$) then                                                    2
        $\text{DECIDED}_c \leftarrow \text{TRUE}$                                    3
        deliver($c, v$)  ▷ Execute requests in $v$ when all earlier decisions executed 4
Upon reception of $t+1$ $\langle\text{ACCEPT}, c, h\rangle$ for hash $h$ do          5
    if Proposal($c$) $= \perp \vee$ hash(Proposal($c$)) $\neq h$ then                6
        send $\langle\text{REQ-DECISION}, c\rangle$ to $2t$ other replicas            7
Upon reception of a $\langle\text{REQ-DECISION}, c\rangle$ from $r'$ do               8
    Create new EventHandler:                                                         9
        Upon isDecided($c$) do                                                       10
            retrieve $\langle c, v, \Gamma\rangle$ from LOG                          11
            send $\langle\text{FWD-DECISION}, c, v, \Gamma\rangle$ to $r'$           12
Upon reception of a $\langle\text{FWD-DECISION}, c, v, \Gamma\rangle$ do              13
    if $\neg$ isDecided($c$) $\wedge$ verify($c, v, \Gamma$) then                     14
        LOG $\leftarrow$ LOG $\cup\{\langle c, v, \Gamma\rangle\}$                   15
        broadcast $\langle\text{FWD-DECISION}, c, v, \Gamma\rangle$ to all           16
        decide($c$,$v$)                                                              17
```

In the subsequent sections, we provide a brief description of this protocol extension for BFT-SMaRt, as outlined in Algorithm 3. The overall pattern of this extension is illustrated in Figure 4.4b.

Requesting a Decision

A correct replica r requests a decision from others for consensus c when it receives $t+1$ $\langle\text{ACCEPT}, c, h\rangle$ (or COMMIT in PBFT parlance) for some hash h *and* it has not yet received a leader proposal ($\perp$) or the received proposal does not match h (Lines 5–7).

It is important to note that a malicious leader can isolate replica r by sending a corrupt or deviating proposal. To address this concern, the requirement ensures that a decision forwarding request is eventually sent only when two conditions are both met:

1. A decision is made by at least a single correct replica (note that this causes r to eventually receive at least $t+1$ ACCEPTs from correct replicas) *and*

2. r was isolated by the leader (that means r did not receive a well-formed proposal matching the digest in the $t+1$ ACCEPTs)

It is worth mentioning that under a correct leader, it is also possible that a decision query is sent if the proposal reaches replica r *late* (for instance assume a slow network link). However, this does not pose a problem as requesting a decision does not block replica r from participating in the normal protocol: r will send its vote messages (WRITE and ACCEPT) regardless;

Responding to a Decision Request

If a correct replica r' receives a forwarding decision request from r, it will promptly answer r once a decision is made (Lines 8-12). Additionally, it is essential for r to query a sufficient number of other replicas to ensure contact with at least one of the $t+1$ correct replicas that are certain to reach a decision eventually. While in our solution a replica contacts these replicas simultaneously, it would be also feasible to query only a single replica at a time to avoid overhead and choose another replica if no response can be obtained.

Furthermore, in our proposed approach, replicas also retain information about which consensus decisions they have forwarded (not demonstrated in the simplified pseudocode). This ensures that decisions can only be requested once per replica and consensus.

Using a Forwarded Decision to Decide Consensus

Lastly, when a forwarded decision reaches r, then r initially verifies if it is still undecided in consensus c to guarantee the *Integrity* property of consensus. Subsequently, r employs the proof Γ to validate the correctness (*Agreement*) of the received consensus decision (Lines 13–14).

The decision is then added to the LOG (Line 15) and disseminated to all other replicas (Line 16). This dissemination, in principle an "echo" phase, guarantees the fulfillment of the *Termination* property of consensus, even if a correct replica decides based on a forwarded decision instead of further participating in the normal phase protocol.

Termination is now simple to prove under this protocol modification: If a single correct replica r decides using a forwarded decision message d, then all correct replicas will also receive d (as it is echoed by r before calling DECIDE in Line 16) and eventually decide d. Without the echo, other replicas might e.g., fail to form a quorum of $2t + 1$ ACCEPT because the correct replica r can skip the agreement protocol after receiving a forwarded decision thus not broadcasting vote messages (WRITE and ACCEPT) for v.

4.3.3 Implications on State Transfer

The protocol outlined in the preceding section operates under the assumption that decisions and their proofs are permanently stored in case a replica requires them to respond to a REQ-DECISION message. PBFT and BFT-SMaRt both come with inbuilt logging capabilities that can be expanded to encompass the logging of this information as well.

Garbage Collection of Log Entries

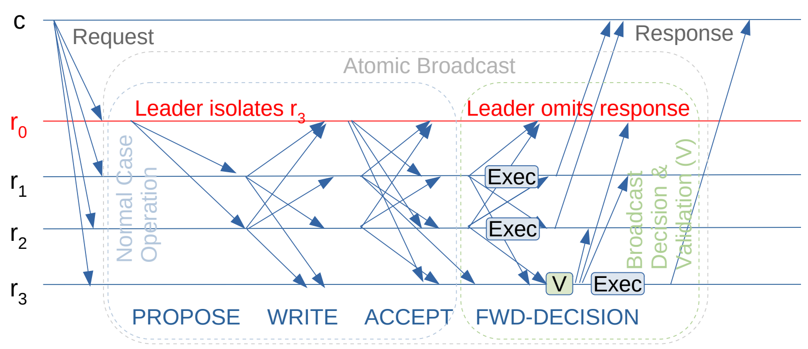
However, to maintain practicality with respect to bounded memory available on the replicas, it is necessary to prevent these logs from expanding indefinitely. To address this, PBFT and BFT-SMaRt employ a garbage collection mechanism that generates state checkpoints and deletes all log entries associated with decisions that were made before the last *stable* checkpoint (for which proof that sufficiently many replicas are up to date exists).

If a replica receives a forwarding request for a decision that has already been garbage-collected, it replies with an error message (OUTDATED-REQ). In this situation, the requesting replica has fallen behind significantly and must initiate a state transfer to synchronize with the remaining replicas. The error message should contain proof of the highest decided consensus to convince the requester.

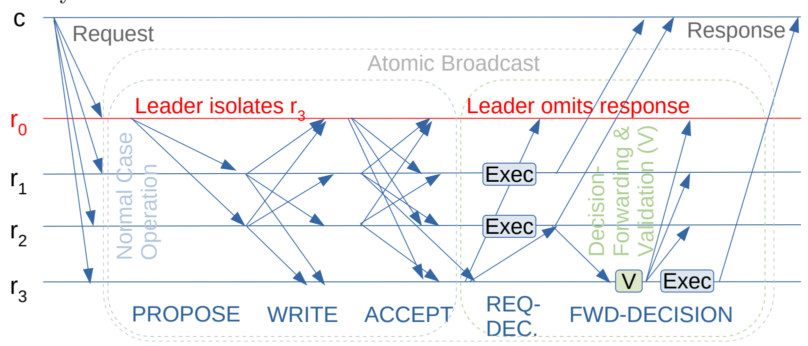
Replies become Part of the Replica State

When employing the read-only optimization in conjunction with our solution, it becomes necessary to augment the replica state with the most recent response sent to each client. This assumes that clients operate in a closed loop, with a maximum of one pending request at a time. If a client can have multiple pending requests (up to k), the state must store the last k responses sent to the client. During the state transfer process, it is crucial to include these responses to ensure that the recovered replica can effectively respond to any pending requests initiated by any correct client. These replies need to be also transferred during state transfer to ensure the recovered replica can reply to outstanding requests issued by any correct client.

If a reply must be re-transmitted due to a client timeout, the replicas that obtained their state from the checkpoint, assuming the checkpoint does not include a reply store, will not be able to respond to the skipped requests (Distler 2021). However, if the latest client replies are transferred



(a) Decision Broadcasting: the normal case operation pattern is extended by an additional all-to-all broadcast of FWD-DECISION.



(b) Decision Forwarding: the normal case operation pattern is extended *on demand* by REQ-DECISION and FWD-DECISION.

Figure 4.4: Correct replicas can propagate a decision and externally-verifiable proof Γ to possibly isolated replicas.

during state transfer, then there will eventually be $2t+1$ correct replicas capable of responding to clients.

4.3.4 Correctness Arguments for the Protocol Adjustments

In this section, we explain the correctness arguments for both proposed protocol modifications. These arguments are based on BFT-SMaRt, but can be adapted for PBFT.

Correctness of Decision Broadcasting

We first show the correct protocol behavior when broadcasting decisions after consensus was reached.

Theorem 4.3.1 **Safety of decision broadcasting**

If a decision is made, then it is the same in all correct replicas.

Proof. This is because a decision is either made by the normal execution pattern in which a quorum Q of $\lceil \frac{n+t+1}{2} \rceil$ received ACCEPTs are used for some value v, or, by validating a forwarded decided value v' using an externally verifiable proof Γ, which contains a quorum Q' of signed ACCEPT messages from replicas supporting v'. Since $|Q \cap Q'| \geq t+1$ holds, assuming that $v \neq v'$ would imply at least one *correct* replica supported v in Q and v' in Q', which is a contradiction because correct replicas never equivocate, so $v' = v$ must hold. □

Before we continue with the liveness argument, we note that both PBFT and BFT-SMaRt guarantee that a request is eventually executed in at least $t+1$ correct replicas. In cases in which a

Byzantine leader reigns and less than $t + 1$ correct replicas decide a value, we know that at least $t + 1$ correct replicas stay undecided. After a timeout, these replicas trigger a synchronization phase in BFT-SMaRt (or a view change in PBFT).

In the synchronization phase, replicas synchronize their logs and reach a consistent state. Note that this procedure might repeat (e.g., consider the possibility of t cascading leader failures) until a stable leader takes over and ensures a decision is reached in at least $t + 1$ correct replicas.

Theorem 4.3.2 **Liveness of decision broadcasting**

A request issued by a correct client will be eventually executed by all correct replicas.

Proof. We first show that if a value is decided in some correct replica, then it is eventually decided in *all* correct replicas.

We assume that the value v was decided in some correct replica r for consensus instance c. Generally, a decision can be made by either gathering enough ACCEPT messages for a prepared proposal v *or* by receiving a $\langle \text{FWD-DECISION}, c, v, \Gamma \rangle$ message with valid proof Γ. In both of these cases, upon deciding v for consensus instance c, replica r broadcasts the $\langle \text{FWD-DECISION}, c, v, \Gamma \rangle$ message to all other replicas, thus enabling all correct replicas to eventually decide v (either during normal case operation pattern or after receiving this forwarded decision). In particular, the fact that any correct replica that utilizes a $\langle \text{FWD-DECISION}, c, v, \Gamma \rangle$ message to decide will echo this message to all other replicas preserves the *Termination* property of the consensus primitive.

Until now, we only showed that a decision made in a single correct replica eventually propagates to all correct replicas. To show that an operation o that needs to be ordered is eventually executed in *all* correct replicas, we suggest using the above result in combination with the liveness proof of the Mod-SMaRt algorithm in (Sousa and Bessani 2012) (Appendix, Theorem 2, using Lemmata 1-3). The Mod-SMaRt algorithm has specific behavioral properties that (i) an operation received in a single correct replica is eventually received in all correct replicas, (ii) eventually (after at most t regency changes after GST), a leader successfully drives progress in the protocol, and (iii) if a single correct replica starts consensus, eventually consensus starts in at least $n - t$ replicas.

Using these properties, Sousa and Bessani (Sousa and Bessani 2012) proved that an operation is eventually decided and executed in at least $t + 1$ correct replicas. This result combined with the decision broadcasting modification ensures that an operation executed by a correct replica is executed by all correct replicas. □

Correctness of Decision Forwarding

We now show the correct protocol behavior when the decision forwarding technique is used.

Theorem 4.3.3 **Safety of decision forwarding**

If a decision is made, then it is the same in all correct replicas.

Proof. Analogous to the safety argument for decision broadcasting. □

Theorem 4.3.4 **Liveness of decision forwarding**

A request issued by a correct client will be eventually executed by all correct replicas.

Proof. We will show that if a value v is decided in *at least one* correct replica, then it is eventually decided in *all* correct replicas. This result can be combined with the liveness proof of the Mod-SMaRt algorithm in (Sousa and Bessani 2012) (Appendix, Theorem 2), which shows any operation o that needs to be ordered is eventually executed in at least $t + 1$ correct replicas.

For the start of our proof, we assume that the value v was decided in some correct replica r in consensus instance c. When using the decision forwarding extension, a decision can be made by either gathering enough ACCEPT messages for a prepared proposal v *or* by receiving a $\langle$FWD-DECISION$, c, v, \Gamma\rangle$ message with valid proof Γ, so we need to consider both cases.

1. *(Case 1) Forwarded Decision:* The decision v was made in r by using a received, valid $\langle$FWD-DECISION$, c, v, \Gamma\rangle$ message. In this case, since replica r decided v using a forwarded decision, it will broadcast $\langle$FWD-DECISION$, c, v, \Gamma\rangle$ to all other replicas, which implies all correct replicas can eventually decide v.

2. *(Case 2) Normal Operation Pattern:* The decision v was made in r by following the normal operation pattern. In this case, we need to first consider how many correct replicas received a PROPOSE for v in consensus instance c:

   (Case 2.a) A PROPOSE for v is received by *all* correct replicas: the concrete behavior depends on the order of messages received:

   - *(Case 2.a.i)* If the proposal is delayed, and some correct replica r' receives $t+1$ ACCEPT messages before it receives the leader's proposal, then it sends a $\langle$REQ-DECISION$, c\rangle$ message to $2t$ other replicas. If r' receives a valid $\langle$FWD-DECISION$, c, v, \Gamma\rangle$ message before completing the normal case operation pattern and uses the forwarded decision to decide, then it broadcasts the $\langle$FWD-DECISION$, c, v, \Gamma\rangle$ to all, thus enabling all correct replicas to eventually decide. Note that *asking* for a decision does not block r' from further participating in the agreement protocol.

   - *(Case 2.a.ii)* If no correct replica sent a forwarding request, it means all correct replicas executed the three-step agreement protocol for c and decided.

   (Case 2.b) There is a correct replica r' that did not receive a PROPOSE for v in c. Here, we need to consider how many correct replicas decided during the normal case operation pattern:

   - *(Case 2.b.i)* If at least $t+1$ correct replicas decide v, r' is guaranteed to have received $t+1$ ACCEPTs from these replicas and thus it sends a $\langle$REQ-DECISION$, c\rangle$ message to $2t$ other replicas. Since the set of $t+1$ correct and decided replicas intersects with any set of $2t+1$ probed replicas (the requester plus the $2t$ requested replicas) in at least one correct and decided replica, this replica will forward its decision and a valid $\langle$FWD-DECISION$, c, v, \Gamma\rangle$ will be received by r'. Then, r' decides v and broadcasts the $\langle$FWD-DECISION$, c, v, \Gamma\rangle$ to all replicas, ensuring all correct replicas eventually decide v.

   - *(Case 2.b.ii)* If less than $t+1$ correct replicas decided during the normal protocol execution, any correct, isolated replica may send a $\langle$REQ-DECISION$, c\rangle$ message to others, depending on whether it received $t+1$ matching ACCEPTs for v (which depends now on Byzantine replicas sending such ACCEPTs). If *any single* correct replica sends a $\langle$REQ-DECISION$, c\rangle$ message *and* receives a valid $\langle$FWD-DECISION$, c, v, \Gamma\rangle$ response, it decides v, and broadcast this decision to all, thus enabling all correct replicas to eventually decide. If this is not the case, then we know at least $t+1$ replicas remain undecided, and will send STOP messages, thus triggering a *synchronization phase* in which replicas synchronize their logs and continue under a new leader. After *GST*, a stable leader will eventually reign after at most t such regency changes.

□

4.4 Implementation in BFT-SMaRt

We implemented our two solutions using the BFT-SMaRt library yielding variants of BFT-SMaRt that we label *decision broadcasting* and *decision forwarding* in the following sections. Besides the incorporation of additional messages to query or send decisions between replicas, these variants are also required to include a ReplyStore object in the replica state that is exchanged during *state transfer*. Further, replicas remember all decisions and their corresponding proofs from their current consensus instance c back to $(c - chkpInt)$ where $chkpInt$ is the configured *checkpoint interval*.

Because we also want to reason about the performance of the system when under attack, we implemented variants that simulate the attack described in Section 4.2.1. In the following, these variants are postfixed with a "*", i.e., *decision broadcasting** and *decision forwarding**. In these variants, the leader is malicious and perpetually conducts the isolation attack.

Moreover, we built a variant of BFT-SMaRt without the read-only optimization. This variant is a very simple modification of the original BFT-SMaRt in which the client collects only a weak certificate of $t + 1$ matching responses instead of a quorum of $\lceil \frac{n+t+1}{2} \rceil$ and any call a client application makes to the invokeUnordered interface is redirected to invokeOrdered instead. This variant allows us to compare the read-optimized variant of BFT-SMaRt with a variant without this optimization to reason about the question if the optimization is worth implementing.

Finally, we created an *integration test*, that captures the scenario of a leader performing the isolation attack at a specific consensus instance c_{attack}. Such a test can be used to validate the system, as it can be easily and automatically checked whether the protocol successfully terminates through all client requests or gets stuck after the instance c_{attack}. The liveness attack that we identified was later also used to validate the capabilities of an automated fuzzing method, ByzzFuzz, that detects fault-tolerance bugs by injecting randomly generated network and process faults into their executions (Winter et al. 2023).

4.5 Evaluation

In this section, we conduct an experimental evaluation of the solutions that we proposed. In our first experiment, we study the benefits of the read-only optimization. Subsequently, we compare the two suggested solutions both in fault-free scenarios and when under attack.

4.5.1 Experimental Setups

For evaluation, we employ two different evaluation setups: for the deployment of the SMR group in either a local area network (LAN) or in a wide area network (WAN).

LAN

To evaluate the performance of our implementations in terms of throughput and latency, we employ a specific configuration: Our experiments involve running 4 replicas on individual Ubuntu 20.04 virtual machines (VMs), each equipped with 4 virtual central processing units (vCPUs) and 10 GB of RAM. These VMs are hosted on an Intel Xeon Silver 4114-based server.

Method. Additionally, we use 4 separate VMs with identical specifications to distribute the clients. For measuring throughput and latency, we employ the micro-benchmarks suite (a performance measuring tool that assumes an empty server implementation) provided by BFT-SMaRt. This tool assesses the throughput at the leader replica, while the latency is measured at a selected client's side.

To comprehensively evaluate system performance, we gradually increase the number of clients participating in the measurements, reaching up to 400 clients. This progressive increase in workload allows us to cautiously approach the performance limits of the deployed system until we observe saturation (at some point launching more clients does not yield an increase in system throughput anymore).

WAN

In our wide-area network experiment, we are especially interested how the protocol adjustments impact the observed latency of the system. This is because the protocol might exhibit more communication steps than usual, which adds to the overall latency in a WAN.

To evaluate the latency performance of a system deployed in a WAN, we use the Amazon AWS cloud infrastructure to deploy multiple EC2 instances across various regions. Since our latency experiments do not require high hardware specifications, we opt for the *t2.micro* instance type. This instance type offers 1 virtual central processing unit (vCPU) and 1 GB of RAM.

Method. For our experiments, we select the following, geographically dispersed regions: *Virginia*, *Ireland*, *São Paulo*, and *Sydney*. In each of these regions, we set up dedicated VMs to deploy clients and $n = 4$ servers, with the leader replica placed in Virginia. The clients send requests simultaneously, and we ensure a sufficient number of clients to achieve a throughput of at least 100 requests per second.

To measure latency, each client samples a total of 1000 requests, with each request having a payload size of 100 bytes. The latency is determined as the time elapsed between a client sending a request and accepting its corresponding execution result. In each region, we have a dedicated measuring client responsible for collecting latency data.

4.5.2 Quantifying the Benefit of Optimized Reads

We first approach a crucial question: *Is it truly advantageous to incorporate the read-only optimization into a BFT protocol?* At least in theory, the optimization offers two potential benefits:

1. Firstly, by allowing read-only requests to bypass the three-step agreement protocol, the number of requests requiring ordering decreases. Consequently, there is reduced coordination necessary between replicas, which has the potential to improve overall system performance.
2. Secondly, in deployments spanning wide areas, the latency between replicas can be significant. Consequently, coordinating actions incurs additional latency costs. By implementing a two-phase request-response pattern instead of the standard five communication steps (including the three extra steps for ordering), it is possible to achieve significantly faster response times for read operations. This becomes especially valuable in scenarios where the optimization helps mitigate the impact of high replica coordination latency.

Evaluation in the LAN Setup

In our initial evaluation, we assessed the performance of (optimized and non-optimized) read and write operations in the LAN setup for a request payload of 4kB, with results shown in Figure 4.5.

Results. From Figure 4.5, it becomes evident that optimized reads consistently deliver lower latencies and increased throughput compared to non-optimized reads. This observation suggests that incorporating the read-only optimization can enhance the performance of a BFT system.

Evaluation in the WAN Setup

Using our WAN configuration, we conducted an experiment to compare the latency benefits of optimized reads with their non-optimized counterparts, which must run a three-step agreement

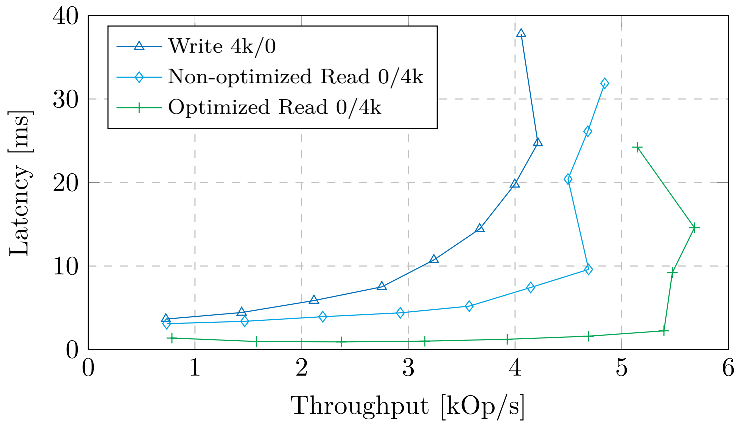


Figure 4.5: Performance comparison between read and write operations in BFT-SMaRt for request / response sizes (in byte) in a LAN.

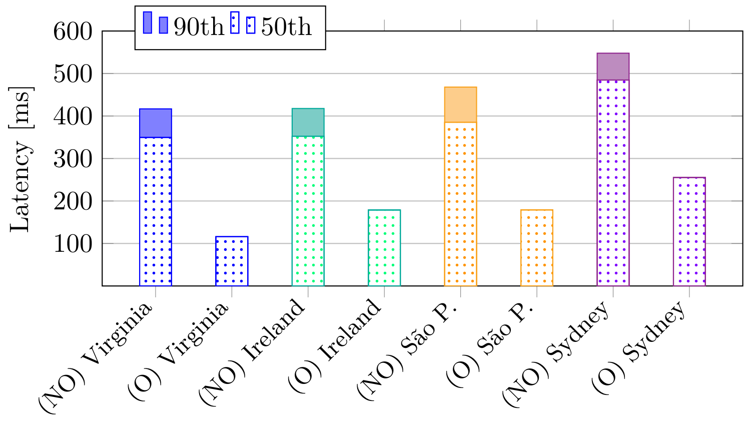


Figure 4.6: Latency comparison between optimized (O) and non-optimized (NO) read operations in BFT-SMaRt deployed in a WAN, as observed by clients in different regions.

protocol. To ensure a fair comparison, we focused on the "opportunity costs", specifically the latency improvements achievable through the incorporation of the optimization.

For non-optimized read operations, we use our non-optimized variant of BFT-SMaRt, where all operations are ordered and clients can accept results as soon as they receive $t+1$ matching replies. In contrast, the optimized variant required a quorum of $q = \lceil \frac{n+t+1}{2} \rceil$ matching replies. Although this distinction is slight, it yields some advantages in terms of latency when operating in a WAN setup. Instead of waiting for the fastest q out of n replicas, the non-optimized approach only waits for the fastest $t+1$ replies.

Results. To quantify these advantages, Figure 4.6 presents the median and 90th percentile of latencies measured from a sample of 1000 requests for a client in each region. On average across all sites, the non-optimized reads exhibit a latency that is 2.16× higher than the optimized variant (median), and this difference increases to 2.54× when considering the 90th percentile. Moreover, non-optimized reads demonstrate a greater variation in observed latency, with the 90th percentile significantly surpassing the median in comparison to optimized reads.

This phenomenon can be attributed to a random waiting period for an ordered request that arrives at the leader. The leader is required to wait for an ongoing consensus instance to complete before it can include the pending request in a batch and propose it. The expected waiting time corresponds to half of the consensus latency on BFT-SMaRt (Berger, Reiser, Sousa, et al. 2022). If we calculate the proportion of latency (averaged across all regions) by dividing the optimized variant through

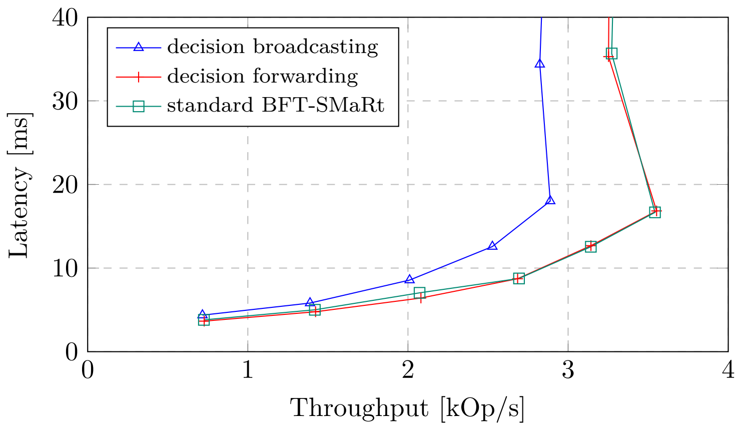


Figure 4.7: Performance comparison in fault-free operation.

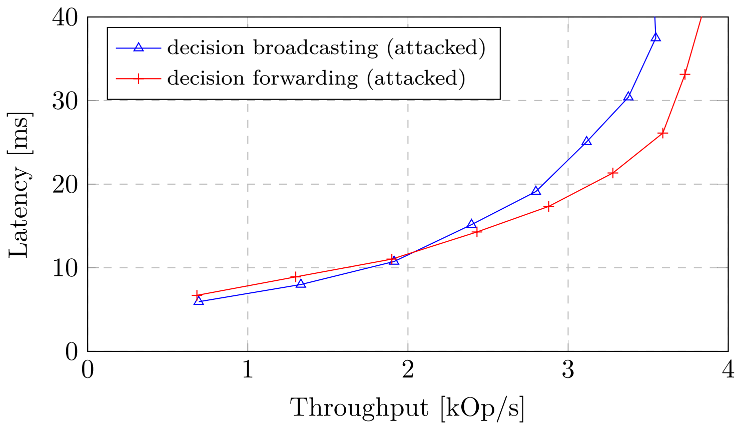


Figure 4.8: Performance comparison when under attack.

the non-optimized variant, it yields 0.457 for the median and 0.389 for the 90th percentile. This observation is interesting when we compare it with the "theoretical reduction" of $\frac{2}{5} = 0.4$, as the read-optimized variant only requires only 2 out of 5 communication steps in the wide-area network, but increases the number of replies a client needs to collect.

Conclusion

Given the evident performance benefits of incorporating the read-only optimization in BFT SMR systems that were deployed in both LAN and WAN setups, we conclude that implementing this optimization in practical BFT SMR systems poses a worthwhile effort.

4.5.3 Comparison of Implemented Solutions

To evaluate the performance impact of implementing both *decision broadcasting* and *decision forwarding*, we examine their effects on the ordering phase. To conduct this investigation, we establish a comparison between the original BFT-SMaRt system (used as a baseline) and our implemented solutions.

Fault-free Execution

To start with, we assess the performance of fault-free execution by using our LAN setup and employing a request/response size of 4kB/4kB for clients. The results obtained from our measurements are depicted in Figure 4.7.

Results. Our findings indicate that the implementation of *decision broadcasting* leads to a notable decrease in performance, whereas *decision forwarding* demonstrates performance levels similar to the original BFT-SMaRt system. These observations align with our expectations:

Broadcasting decisions as an additional phase induces overhead in the system. On the other hand, *decision forwarding* follows the regular message pattern of BFT-SMaRt as long as no decision is requested, which is typically the case under a correct leader. As a direct result, the *decision forwarding* variant can achieve comparable performance.

Failure Scenario

In the subsequent experiment, we study the impact of a Byzantine leader conducting an isolation attack on the system. Initially, we compare the performance of the two solutions *decision broadcasting** and *decision forwarding** variants in our LAN setup when under attack (remember the *-variants denote *under attack* scenarios). The results obtained from this comparison are illustrated in Figure 4.8. It is worth noting that we cannot evaluate the performance of the standard BFT-SMaRt system when under attack, as it would prevent clients from accepting the operation result due to the liveness problem we explained in Section 4.2.1.

Results. In the current scenario, we observe that the performance of *decision forwarding** is much closer to that of *decision broadcasting**, with the former exhibiting only a slight advantage. This outcome can be attributed to the fact that *decision forwarding* always involves probing two replicas to forward a decision, whereas decision broadcasting requires *all* replicas to disseminate the decision.

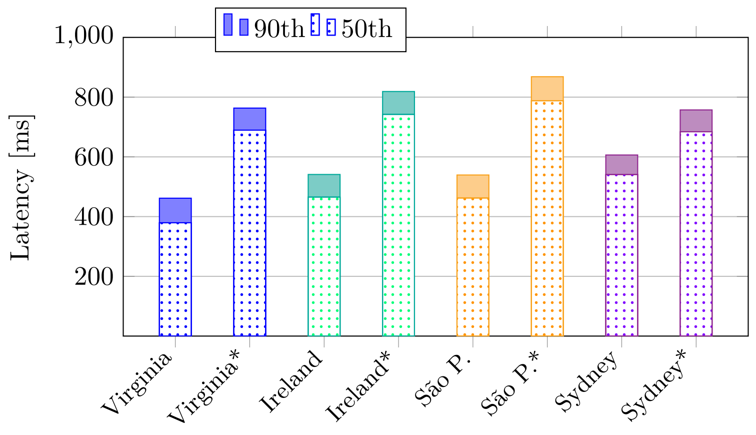
Interestingly, the overall performance improves compared to the fault-free experiment. The reason for performance to improve is, that the attack prevents the leader from proposing to a specific replica and responding to the clients (up to 400 in the system), thereby reducing the workload on the leader. Since the leader is typically the bottleneck, performing less work leads to increased performance. This applies to both *decision broadcasting* and *decision forwarding* when under attack. However, in the case of *decision forwarding**, this effect becomes less evident in the diagram due to the additional work performed by replicas by forwarding decisions, which negatively impacts performance (remember that they do not need to do this in fault-free execution).

Conclusion

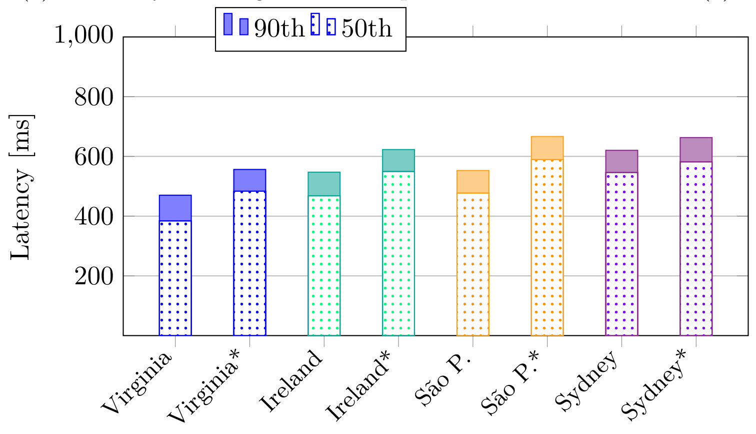
In the LAN environment, *decision forwarding* outperforms *decision broadcasting.* This is due to *decision forwarding*'s ability to avoid unnecessary overhead during fault-free operations, resulting in improved performance. Additionally, *decision forwarding* exhibits better performance during an isolation attack, thus showing an advantage in such scenarios.

Evaluation in the WAN

To assess the implications of our solutions on BFT SMR systems deployed in WAN environments, we conducted evaluations of decision broadcasting and decision forwarding in both fault-free ex-



(a) *decision forwarding* in normal operation and when attacked(*).



(b) *decision broadcasting* in normal operation and when attacked(*).

Figure 4.9: Latency comparison of implemented variants in a WAN deployment as observed by clients located in different regions.

ecutions and when under attack (*-variants). These evaluations were performed using our WAN setup, and the measurement results are presented in Figure 4.9.

Results. A notable distinction can be observed in the impact on latencies when an isolation attack occurs. During an attack, *decision broadcasting* demonstrates superior performance compared to *decision forwarding* across all sites within the system. Specifically, when averaging across all sites, the median latency in decision forwarding increases by 57%, whereas in *decision broadcasting*, it only rises by 18%. This shows the advantage of *decision broadcasting* in mitigating the latency increase during an isolation attack.

The disparity in performance can be attributed to two main factors. Firstly, in *decision broadcasting*, replicas promptly disseminate the decision to all others without any additional communication steps. On the other hand, *decision forwarding* requires an initial REQ-DECISION message, which introduces an extra communication step, thereby adding to the overall latency (refer to Figure 4.9a).

Secondly, replicas do not reach a decision simultaneously, but rather at different times. As all replicas broadcast their decisions, the speed at which the isolated replicas catch up is determined by the fastest replica among them. This side-effect benefits the observed latency of *decision broadcasting* during an isolation attack, as shown in Figure 4.9b.

Conclusion

In WANs, during an attack, *decision broadcasting* offers the advantage of eliminating an additional communication step. In this scenario, the arrival of the fastest broadcast decision sets the pace for isolated replicas to synchronize and catch up. Overall, these factors make *decision broadcasting* an appealing alternate choice for WAN deployments.

4.6 Additional Related Work

In this section, we discuss some additional related research works that focus on optimizing reads in SMR or enhancing (or validating) the robustness of BFT protocols and their implementations.

The variety of these techniques stems from employing different fault models or relaxing consistency guarantees. In our overview of related work, we will mainly focus on SMR.

Optimizing Reads in BFT SMR

Castro and Liskov were the pioneers in introducing the read-only optimization for BFT SMR, which facilitates fast reads without the need for ordering and without compromising linearizability. Their groundbreaking work was initially presented in PBFT (Castro and Liskov 1999). A few other BFT SMR systems followed to also provide a read-only optimization, e.g., Q/U (Abd-El-Malek et al. 2005), HQ (Cowling et al. 2006), PBFT-CS (Wester et al. 2009), UpRight (Clement, Kapritsos, et al. 2009), Zyzzyva (Kotla et al. 2007), and BFT-SMaRt (Bessani, Sousa, et al. 2014) (and its variants (Sousa and Bessani 2015; Berger, Reiser, Sousa, et al. 2022)).

To the best of our knowledge, the issue discussed in Section 4.2.1 does not apply to all of the systems mentioned. When generalizing our insights the vulnerability seems to exist in BFT SMR designs that meet one of the following criteria:

1. A BFT protocol that is agreement-based and employs the same normal-case ordering pattern as PBFT, which fails to guarantee execution in a sufficient number of replicas to ensure that a quorum of replies (necessary for linearizability) is received by the client,
2. A BFT protocol that does not enhance the replica state with the last response sent to each client, resulting in correct replicas being unable to respond to re-transmitted requests after state transfer;

Upon closer examination, it becomes apparent that this vulnerability currently exists in multiple frameworks within the field (e.g., (Castro and Liskov 1999; Cowling et al. 2006; Wester et al. 2009; Bessani, Sousa, et al. 2014; Sousa and Bessani 2015; Berger, Reiser, Sousa, et al. 2019)).

Note that BFT SMR designs exist where the read-only optimization does not induce a vulnerability. An example is Zyzzyva (Kotla et al. 2007), which uses a combination of a forwarding technique (proposals are signed and can be queried from other replicas) and a sub-protocol for blaming a faulty leader. In the scenario where a Byzantine leader isolates a correct replica by transmitting conflicting proposals, the isolated replica can construct a proof of misbehavior (POM). This proof contains signed messages from the leader, exposing their malicious actions, and is then broadcasted. Upon receiving the POM, the replicas initiate a view change to elect a new leader.

It is important to note that, in general, an isolation attack cannot be detected if the malicious leader solely refrains from sending messages. Additionally, in the event of an isolation attack, forwarding decisions is generally considered preferable to forwarding proposals. This choice helps avoid additional latency caused by the subsequent protocol steps.

Optimizing Reads in CFT Systems

In CFT SMR, a commonly employed idea is to let clients read from a single replica, which requires the usage of temporary leases to preserve linearizability under the constraints of this optimization.

The concept of a *leader lease* was introduced by Chandra et al. in their work on Paxos (Chandra, Griesemer, et al. 2007). This concept ensures that a stable Paxos leader possesses the lease, guaranteeing that operations maintain up-to-date information. Consequently, reads can be served by the leader locally, ensuring that they are never stale and satisfying the property of linearizability. This idea has been adopted in several practical systems, such as Spanner (Corbett et al. 2013), which employs “snapshot reads” to enable clients to read locally at specified timestamps, and Chubby (Burrows 2006).

The concept of leases provides a wide range of possibilities for system design. In addition to leader leases, it is also feasible to grant leases to a set of replicas or even to all replicas within the system (Moraru et al. 2014; Baker et al. 2011). For example, Moraru et al. introduced the concept of “Paxos Quorum Leases” (Moraru et al. 2014), where leases for different objects can be managed by individually chosen quorums of replicas. By selecting a Paxos quorum for a lease, the management of leases becomes more efficient, and overhead can be reduced by combining lease revocation and Paxos messages. This approach offers flexibility in managing leases and optimizing the lease management procedure within the system.

ZooKeeper (Hunt et al. 2010) enables fast local reads from any server, providing a level of flexibility in the system. However, it comes with a relaxation in consistency guarantees. While ZooKeeper offers sequential consistency, it does not guarantee linearizability, which means that stale reads can potentially occur within the system. It is important to note that a SMR framework should preserve linearizability of the service implementation (ZooKeeper does not guarantee linearizability).

In contrast to these works, this thesis focuses only on BFT system designs, which are notoriously more complex when it comes to implementing read-only requests than their CFT counterparts.

Optimizing Reads in a Hybrid Fault Model

Troxy (B. Li et al. 2018) leverages trusted execution environments (TEEs) to transfer client functionality, such as output consolidation, to the replica side. This approach enables clients to access the BFT system transparently. Additionally, Troxy introduces a unique read optimization strategy that involves actively managing a *fast-read cache*. This cache reflects the state changes resulting from the latest write operation, ensuring strong consistency guarantees. To ensure the security of the system, Troxy assumes that the functionality within the TEE never exhibits Byzantine behavior. Consequently, replicas are required to be equipped with a TEE, allowing for the effective implementation of the Troxy approach.

Robustness of BFT SMR

Related research also focuses on increasing robustness and closing vulnerabilities of BFT SMR design like Aardvark (Clement, Wong, et al. 2009), Spinning (Veronese et al. 2009), Prime (Amir, Coan, et al. 2010), RBFT (Aublin, Mokhtar, et al. 2013) and BFT-Mencius (Milosevic et al. 2013).

Aardvark (Clement, Wong, et al. 2009) aims to enhance the resilience of BFT systems against performance degradation attacks by introducing a set of design principles. These principles include the use of signed client requests, resource isolation, and regular view changes. These measures help mitigate the impact of attacks on system performance.

Spinning (Veronese et al. 2009) proposes a different approach by allowing replicas to take turns being the leader in the system. This rotation of leadership aims to distribute the workload and increase system resilience.

Prime (Amir, Coan, et al. 2010) introduces novel mechanisms to enhance BFT systems' robustness against various attacks, including denial of service, corrupt message authentication code, and PRE-REPARE-Delay attacks. These mechanisms are designed to detect and mitigate the effects of these attacks, thereby improving system resilience.

RBFT (Aublin, Mokhtar, et al. 2013) suggests running multiple instances of the ordering protocol redundantly to enhance system robustness. This redundancy enables the detection and replacement of a misbehaving leader that performs noticeably slow, thereby maintaining system performance.

SplitBFT (Messadi et al. 2022) strengthens the robustness of BFT SMR (in particular the safety aspect) using trusted compartments. Its idea is to take the core logic of BFT protocols and distribute it into multiple compartments resulting in an overall more resilient architecture.

A recent work studies the liveness and latency of BFT SMR by introducing a modular framework that is based on a view abstraction generated by an SMR *synchronizer* primitive to drive the agreement on command ordering (Bravo et al. 2022a). The formal specification of the SMR synchronizer is used to prove the liveness of the non-optimized algorithmic core of PBFT.

The Twins methodology is a recent approach for validating BFT protocols (Bano, Sonnino, Chursin, et al. 2022): Essentially, Twins is an unit test case generator that can simulate Byzantine attacks by duplicating cryptographic identities of replicas (which then leads to forgotten protocol states, or equivocations) and can be employed in a simulator to explore a variety of attacking scenarios, possibly detecting safety violations in vulnerable BFT protocol designs.

Recently, a method for liveness checking of the HotStuff protocol family has been proposed (Decouchant, Ozkan, et al. 2023): In their approach, the authors first define *partial state* and *hot state* for capturing information on HotStuff executions, then apply temperature checking and lasso detection to dectect repeating states which are a symptom of potential liveness violations.

Moreover, another recent work focuses on randomized testing of Byzantine fault-tolerant algorithms by introducing ByzFuzz (Winter et al. 2023), a fuzzing method to automatically spot bugs in implementations of BFT SMR protocols. The authors of ByzFuzz showed that using their fuzzing method, they can reproduce the vulnerability we analyzed in Section 4.2.1 in an automated way.

To summarize the above works, the research on how to strengthen the robustness of BFT design is a continual and enduring journey. This thesis contributes to this endeavor by addressing an existing deficit in the PBFT design, a BFT protocol that was proposed over 20 years ago and is widely recognized. This shows how quickly optimizations can invalidate system properties and that it may not be sufficient to prove the intended properties only for the base (non-optimized) system.

4.7 Concluding Remarks

Initially, we explained an existing vulnerability in the design of BFT SMR systems, specifically in PBFT (Castro and Liskov 1999) and BFT-SMaRt (Bessani, Sousa, et al. 2014).

This vulnerability is caused by an optimization technique called *read-only operations* which was first introduced in PBFT (Castro and Liskov 1999). To maintain linearizability when employing this optimization, clients must collect a larger number (i.e., a quorum) of replies. We showed an attack scenario in which a Byzantine leader isolates a maximum of t correct replicas, effectively obstructing clients from attaining this required quorum of replies, and, ultimately resulting in a disruption of the system's liveness.

Interestingly, insights from this thesis influenced the field of validation of BFT protocols. For instance, ByzFuzz (Winter et al. 2023) is a fuzzing method that could later reproduce the vulnerability we found in an automatically generated test scenario.

Next, we explained two solutions, *decision broadcasting* and *decision forwarding*. By employing either one of these solutions, we guarantee *all correct replicas can respond to client requests* and facilitate the liveness of the system even when using the read-only optimization. These solutions allow for two-step read operations while maintaining linearizability and liveness within the system.

Another important insight is that when using the read-only optimization, *state transfer* must ensure that the transferred state includes the most recent replies sent to each client. This is essential to guarantee that recovered replicas are capable of responding to pending requests issued by any client.

As a direct result of this thesis, our patch is open-source and available as a contribution to the official BFT-SMaRt (Bessani, Sousa, et al. 2014) repository[3], a framework that is widely used by the BFT research community.

Evaluation results show the benefits associated with the read-only optimization, along with the performance of our implemented variants, namely *decision broadcasting* and *decision forwarding*, in both LAN and WAN setups. The evaluation results indicate that optimized reads increase overall system performance. In a WAN setup, the optimized read systems exhibit significantly faster latencies compared to their non-optimized counterparts, with non-optimized read latencies being 2.16 to 2.54 times higher (median and 90th percentile respectively).

When comparing the performance of *decision broadcasting* and *decision forwarding*, we observe that the forwarding technique outperforms decision broadcasting due to its minimal overhead, making it the preferable choice for replicas located close to each other, such as within a data center. However, in a WAN scenario, decision broadcasting achieves better latencies than forwarding during the failure scenario.

[3]See `https://github.com/bft-smart/library`

Chapter 5

Adaptive Wide-Area Replication

In recent years, there has been growing academic interest in BFT SMR protocols due to their suggested use in blockchain infrastructures. A prominent example is the BFT-SMaRt library, which was used as an ordering service for the Hyperledger Fabric (HLF) blockchain platform (Sousa, Bessani, and Vukolić 2018). By employing the WHEAT approach (see Section 3.4) with HLF, this combination enables a high-performance and resilient ordering of transactions with sub-second latencies even in geographically distributed environments.

When it comes to practical deployments, there is a big difference between deploying a replica group in a single data center or distributing replicas across different *regions* in the world. In a single data center, latencies between servers are generally relatively consistent across all pairs of servers. Yet in wide-area network environments, we observe *high variations* in latency of communication links connecting different regions (see Figure 5.1). When protocols are waiting for a set of messages to be received, the speed at which a replica makes progress is determined by the slowest replica within a subset of replicas that forms a *quorum* (see Section 2.1.3).

	Oregon	Ireland	Sydney	São Paulo	Virginia
Oregon		68	69	93	40
Ireland	68		133	92	35
Sydney	69	133		157	99
São Paulo	93	92	157		70
Virginia	40	35	99	70	

Figure 5.1: Example: Heterogenous communication speeds (in milliseconds, one-way) in a WAN.

Through the use of additional spare replicas and *weighted replication*, WHEAT demonstrated that the system can benefit from *increased flexibility* in respect to *quorum formation rules* in a wide are network. Consequently, if a system can make progress by accessing a proportionally smaller quorum of well-connected replicas, it leads to a significant reduction in observed latency.

The need for adaptivity in weighted replication. However, reaping the benefits of weighted replication relies on selecting a near-optimal weight configuration. In practice, manually determining the best system configuration for a given set of replicas (and their locations) is challenging:

1. The configuration space is vast as there are many possibilities of distributing the weights between the replicas in the system,
2. To assess the performance of a configuration it requires knowledge of network characteristics *and* quorum formation rules within the BFT SMR protocol behavior;

Moreover, network characteristics can undergo variations during runtime, which means that the optimal configuration may also change dynamically. The optimal configuration may change also due to *failures* of individual replicas. To make weighted replication practical and usable, a self-optimization mechanism is necessary within the SMR system that allows the system to adapt to its environment and optimize the system's consensus latency.

This chapter covers insights that have been partly published in two earlier research articles:

Christian Berger, Hans P. Reiser, João Sousa, and Alysson Bessani (2019). “Resilient Wide-Area Byzantine Consensus Using Adaptive Weighted Replication”. In: *Proceedings of the 38th IEEE Symposium on Reliable Distributed Systems (SRDS)*,

Christian Berger, Hans P. Reiser, João Sousa, and Alysson Bessani (2022). “AWARE: Adaptive Wide-Area Replication for Fast and Resilient Byzantine Consensus”. In: *IEEE Transactions on Dependable and Secure Computing* 19.3, pp. 1605–1620. DOI: 10.1109/TDSC.2020.3030605,

In this chapter, we present a mechanism for geo-replicated state machines called **A**daptive **W**ide-**A**rea **RE**plication (AWARE).

AWARE enables geo-replicated state machines to autonomously optimize themselves during runtime (see Figure 5.6). Our approach involves continuous self-measurements of the communication link latencies between replicas and analyzing the expected consensus latency of the leader based on different weight distributions and leaders. AWARE uses a prediction model of the BFT algorithm to select a configuration that reduces the consensus latency. By reconfiguring the system to minimize consensus latency based on collected measurements and the model’s calculation, the overall mechanism leads to observable latency improvements for clients distributed worldwide.

5.1 System Model

To begin with, we present our system model, which is based on the system model used in WHEAT. We provide a brief overview of this model in this section.

Deployment model. The system consists of a *fixed-sized* set of n replicas denoted by $\mathcal{R} = \{r_0, .., r_{n-1}\}$, which are deployed within a set of $m \leq n$ regions $\mathcal{L} = \{l_0, .., l_{m-1}\}$. The function $l(r) : \mathcal{R} \rightarrow \mathcal{L}$ maps a replica r to the region it has been deployed in. Initially, a system administrator has to select the number of spare replicas Δ such that $0 \leq \Delta \leq n - 4$. Further, we assume the existence of an arbitrary large set of clients $\mathcal{C}$ that can submit requests to the replicas.

Fault model. Our fault model assumes a *threshold adversary* (see Section 2.1.2). The adversary controls a threshold of t replicas which the adversary may choose freely from the replica group. The parameter t is the bound on how many replicas are assumed to be Byzantine at most and needs to be carefully chosen by the system designer prior to the system start, such that

$$n = 3t + 1 + \Delta \tag{5.1}$$

holds, with $\Delta \geq 0$ being the extra spare replicas provided to the system. This means that the adversary controls less than a third of the replicas regardless of how Δ is chosen. It also means that the resilience of the system (specifically: the proportion of replicas that is “allowed to fail”) depends on how a system administrator chooses Δ, resulting in an interesting *trade-off design choice* between resilience threshold and system performance before the launch of the system. We will explore this trade-off situation in more detail later.

Moreover, we adopt the *Byzantine fault model* to describe the behavior of faulty replicas. Replicas under the control of an adversary may freely deviate from the protocol description and even collude in a malicious way, but are still bound by computational capabilities. For example, the attacker cannot break strong cryptographic primitives to bypass authentication methods.

Communication model. To ensure liveness, we employ the partially synchronous system model, as defined in (Dwork et al. 1988). This model is also popular in other works on SMR like PBFT (Castro and Liskov 1999), HotStuff (Yin et al. 2019), MirBFT (Stathakopoulou et al. 2022), and Kauri (Neiheiser, Matos, et al. 2021). In particular, we assume the GST flavor of a partially synchronous system (see Section 2.1.1):

Initially, the system can operate in an asynchronous manner, but we assume the existence of an arbitrary upper bound δ on communication delays after an unknown global stabilization time (GST). Only when GST is reached, we can ensure that the system continues to make progress. It is important to note that the system always maintains safety, even during periods of asynchrony.

Communication channels are point-to-point, authenticated, and reliable. We achieve this by employing TLS-secured channels over TCP/IP. These channels are established using public key pairs that are generated and distributed before the system start time. Each replica $i \in \mathcal{R}$ is assumed to possess an RSA key pair (sk_i, pk_i) and knows all public keys linked to the identities of all replicas.

We also assume that the attacker can control and potentially disrupt the entire network, but only for a limited duration. Once a sufficient level of synchrony is achieved (modeled by GST), the system is guaranteed to make progress again.

5.2 Monitoring Strategy

The self-optimization approach introduced by AWARE demands reliable measurements of communication link latencies as a decision-making basis. In this context, *reliable* means all correct replicas will eventually see the same measurements at some specified logical point in the protocol execution, and the influence of Byzantine replicas in distorting the measurement procedure is limited. In this section, we explain the monitoring strategy of AWARE that leads to reliable measurements of communication links between all replicas, resulting in a $n \times n$ matrix M (e.g., see Figure 5.4) that is consistent in all correct replicas after every completed consensus instance in BFT-SMaRt.

5.2.1 Monitoring BFT Agreements

To begin with, we first showcase potential problems that might arise when monitoring BFT agreements is done in a naive way.

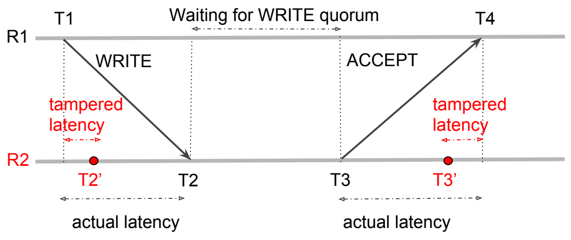


Figure 5.2: Problem with timestamps and Byzantine replicas.

Naive approach. Suppose we would let replicas measure sent and reception times of agreement messages (like PROPOSE, WRITE and ACCEPT) and then include these as *timestamps* in subsequent agreement messages. We could let replicas calculate the one-way latency using timestamps generated by both parties, i.e., as an estimate of half of the round trip time (RTT) between two replicas using the formula:

$$lat_{oneway} = \frac{(T_4 - T_1) - (T_3 - T_2)}{2} \tag{5.2}$$

as shown in Figure 5.2. However, there are three potential problems with this approach:

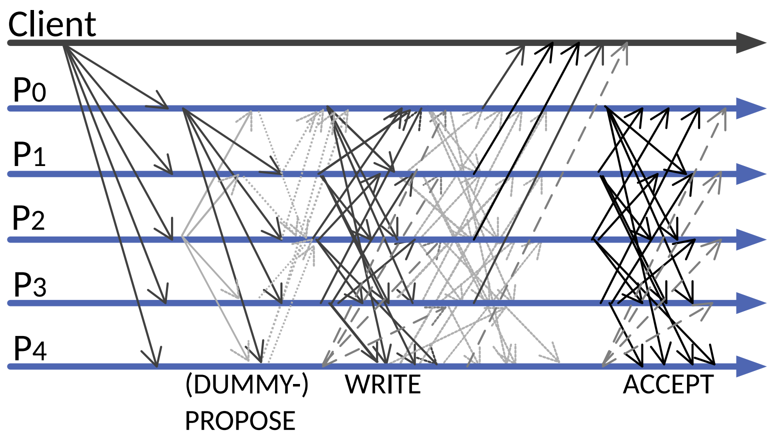


Figure 5.3: Message flow of AWARE ($t = 1; \Delta = 1$), the monitoring messages are shaded.

1. There is *no causal dependence* between the ACCEPT and WRITE message. Replica r_2 might form a WRITE quorum without r_1 and the WRITE from r_1 might even arrive at r_2 after r_2's ACCEPT arrives at r_1.
2. Byzantine replicas can *forge timestamps*. Imagine a malicious replica r_2 might try to shift T_2 closer to T_1 and T_3 closer to T_4, while correct replica r_1 has no means to detect this lying behavior and thus attributes r_2 a better link latency (as shown in the Figure 5.2).
3. Within the replica group, there may be *inconsistent views* towards these measurements.

By this example, we see that we need to account for a few subtleties to guarantee that the self-monitoring procedure can produce reliable and consistent measurements in all correct replicas.

The AWARE Approach.

To deal with these subtleties, we approach the monitoring procedure by applying a set of ideas that both prevent the manipulation of measurements (at least to a large extent) and ensure consistent views on measurements in all correct replicas.

One-sided measurements. A replica sends a *probe* message to another replica that needs to immediately respond with a *probe-reply* message. The response needs to include the same unique, random number that the probing replica included initially in the probe message to prevent replies are sent ahead of time. The one-way latency is calculated as half of this RTT. This guarantees that a measurement taken by a correct replica is not tampered with, i.e., a Byzantine replica can only make itself appear slower than normal but not faster. Since measurements are taken on the side of the measuring replica, there is no need to include any timestamps in the messages.

Probe messages are finally realized using the existing protocol messages, PROPOSE and WRITE. For the response messages, we introduce new messages: PROPOSE-RESPONSE and WRITE-RESPONSE.

The differentiation between a PROPOSE and WRITE message is explained by the fact that proposals are usually larger than the WRITE (and ACCEPT) messages (which only contain the cryptographic hash of the proposal). Thus, proposals may take more time to transmit.

Proposal time of non-leaders. Estimating the propose latency is relevant to optimize the leader location. AWARE also measures the propose latencies of redundant replicas (non-leaders) proposing to other replicas to determine hypothetical latency reductions for the system when using a different replica as the protocol leader. This approach is somewhat similar to (Aublin, Mokhtar, et al. 2013), where redundant agreement instances are observed as a whole (in different leader configurations).

To prevent excessive overhead and avoid compromising the system's performance, non-leaders refrain from simultaneous proposing. This approach aligns with our objective of enhancing performance. Instead, we employ a *rotation scheme* where only one additional replica concurrently broadcasts a DUMMY-PROPOSE alongside the leader. This replica proposes a dummy batch in the same manner as the leader would but does not initiate a new consensus instance. All replicas disregard this proposal. Replicas respond with a PROPOSE-RESPONSE, incorporating the proposed batch in the response message to maintain symmetric behavior concerning communication link latency. Note that the utilization of the DUMMY-PROPOSE is optional since it introduces overhead to the system (refer to Section 5.6.5). An alternative approach involves estimating these latencies by leveraging the measurements AWARE obtains through WRITE-RESPONSE messages only.

Figure 5.3 shows the *message flow* of AWARE with monitoring messages illustrated shaded. Note that this figure also shows the optional optimization of *tentative executions* (see Section 2.3.2). The displayed variant of AWARE obtains the highest accuracy in leader selection as it uses both PROPOSE-RESPONSE and WRITE-RESPONSE messages.

Synchronisation of measurements and actions. AWARE determines the number of recent monitoring messages to be considered in a configurable *monitoring window*. Each replica $i \in \mathcal{R}$ maintains a latency vector L_i which represents connection speeds to the other replicas, which is computed by taking the median over all last measured latencies as defined by the monitoring window.

Furthermore, each correct replica $i \in \mathcal{R}$ periodically shares its latency vector L_i (representing its observations) with all n replicas. This dissemination of measurements occurs at predefined synchronization intervals using the total order broadcast interface of the protocol. This means, all measurement messages pass through the consensus layer and are ordered alongside regular client requests, thus they appear ordered in the decision log. Consequently, for each of the two types of measurements (either PROPOSE or WRITE latency) all replicas obtain some latency matrix M that is consistent at a specific consensus[1]:

$$M = \begin{bmatrix} L_0 \\ \vdots \\ L_{n-1} \end{bmatrix} = \begin{bmatrix} \sigma_{0,0} & \cdots & \sigma_{0,n-1} \\ \vdots & \ddots & \vdots \\ \sigma_{n-1,0} & \cdots & \sigma_{n-1,n-1} \end{bmatrix} \tag{5.3}$$

This results in two $n \times n$ latency matrices M^P and M^W which monitor PROPOSE and WRITE latencies between all replicas, respectively. To make reconfiguration decisions, we employ a deterministic procedure, utilizing the same monitoring data across all correct replicas as a decision-making basis. This procedure ensures that uniform decisions on actions (i.e., how to optimize the system) will be possible assuming replicas employ a deterministic algorithm to determine an action. As soon as replicas synchronized their measurements for a specific consensus instance, they use a *prediction model* that we explain in more detail in Section 5.3.2 to optimize weight distribution and leader. Actions are triggered after each *calculation interval*, that defines the number of consensus instances after which a performance prediction based on the updated monitoring data is conducted.

Bounding monitoring overhead. Using a configuration option for *monitoring overhead* $\omega \in [0,1]$, AWARE can reduce the monitoring overhead if desired. This parameter tunes overhead caused by the monitoring messages. Implementation-level aspects, such as the frequency of sending specific monitoring messages like DUMMY-PROPOSE, are automatically derived based on the concrete value of ω. Higher frequencies of measurements result in more up-to-date monitoring data and enable quicker responses to environmental changes. However, it's important to note that too frequent measurements can also have a detrimental effect on the system's maximum throughput.

[1]This means that when it is accessed for a specific index i in the decision log, then it reflects all measurements recorded in the decisions up to i.

5.2.2 Sanitization of Measurements

In AWARE, the replica state is enriched by two objects to store synchronized measurement information. These objects are $n \times n$ latency matrices M^P and M^W which hold the measurements of PROPOSE and WRITE latencies between all replicas respectively, both initially filled with entries

$$M^{\Box}[i,j] \leftarrow \begin{cases} +\infty, & \text{if } i \neq j \\ 0, & \text{otherwise} \end{cases} \tag{5.4}$$

$M^{\Box}[i,j]$ expresses the latency of replica i to j as measured by i for message type $\Box$ (either PROPOSE or WRITE). Further, replica i can update a row of these matrices with its measurements L_i^P and L_i^W. Using the INVOKE interface of BFT-SMaRt, replica i can update a single row of these matrices (at position i) with its own measurements L_i^P and L_i^W as follows:

$$\textit{invokeOrdered}(\langle \text{MEASURE}, L_i^P, L_i^W \rangle);$$

This interface ensures a global total order on all requests and measurement messages. During the updating procedure, the matrix $M^{\Box}$ is modified, where $M^{\Box}[i,j]$ is set to the last delivered $L_i^{\Box}[j]$ if replica i sent its measurements within the last calculation interval c of measurement rounds. If replica i did not send any measurements within the last c rounds, the corresponding entry is marked with the missing value $(+\infty)$. This approach ensures that for any specific consensus instance, all correct replicas have a consistent view of the measurements. Note that the state transfer needs to also transmit monitoring information (the snapshot of both matrices at the time a checkpoint is taken) in case a recovered replica needs it.

Before using these matrices, we sanitize them to minimize the impact of any malicious replicas. We do this by exploiting two ideas (Figure 5.4 presents an example for sanitization):

- *Link symmetry*: We exploit the symmetry characteristic of communication latency and let replicas have a pessimistic standpoint on measurements. They use the pairwise larger delay in calculations, so that correct replicas can correct underreported latency from Byzantine replicas. This procedure yields

$$\bar{M}^{\Box}[i,j] \leftarrow max(M^{\Box}[i,j], M^{\Box}[j,i]) \tag{5.5}$$

- *Lower bound of links*: Even Byzantine replicas cannot transmit information to each other faster than the speed of light. We use the known locations $\mathcal{L}$ of replicas (given by $l(r)$ for replica r) to calculate the minimum latency[2] for light traveling from $l(i)$ to $l(j)$ over the shortest spherical distance (orthodrome) using the haversine formula (given by $lightLatency(from, to)$). This procedure yields

$$\hat{M}^{\Box}[i,j] \leftarrow \begin{cases} \bar{M}^{\Box}[i,j], & \text{if } \bar{M}^{\Box}[i,j] \geq lightLatency(l(i),l(j))/\frac{2}{3} \\ lightLatency(l(i),l(j))/\frac{2}{3}, & \text{otherwise} \end{cases} \tag{5.6}$$

The final, sanitized latency matrix $\hat{M}$ enjoys the following benefits: Because of the symmetry rule, Byzantine replicas cannot fraudulently improve their link latency to any *correct* replica. Moreover, a Byzantine replica cannot denounce (worsen a link latency to) a correct replica without being contributed a bad latency itself. This is not harmful for the optimization, because ideally we would not want to let the quorum formation speed depend on a slow link in which a Byzantine replica is involved anyways. In the event of $f > 1$, a group of Byzantine replicas may display

[2] Note that $\frac{2}{3}$ of vacuum lightspeed is accepted as the upper bound on data transmission speed for the internet (Cangialosi et al. 2015; Kohls and Diaz 2022) as light travels slower in optical fiber. We consider this limitation when computing the fastest possible network link latency between two locations.

$$\begin{bmatrix} 0 & 68 & 69 & 93 & 40 \\ 67 & 0 & 133 & 92 & 35 \\ 69 & 132 & 0 & 157 & 99 \\ 92 & 92 & 156 & 0 & 70 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(a) Before sanitization.

$$\begin{bmatrix} 0 & 68 & 69 & 93 & 40 \\ 68 & 0 & 133 & 92 & 35 \\ 69 & 133 & 0 & 157 & 99 \\ 93 & 92 & 157 & 0 & 70 \\ 40 & 35 & 99 & 70 & 0 \end{bmatrix}$$

(b) After sanitization.

Figure 5.4: Sanitization of a latency matrix M. Here: red is faulty.

```java
public static double calculateSphericalDistance(double lat1, double lon1, double
    lat2, double lon2) {
    // Convert to radians
    double phi1 = Math.toRadians(lat1);
    double phi2 = Math.toRadians(lat2);
    double deltaPhi = Math.toRadians(lat2 - lat1);
    double deltaLambda = Math.toRadians(lon2 - lon1);


    // Calculate the distance using the haversine formula
    double a = Math.sin(deltaPhi/2) * Math.sin(deltaPhi/2) +
    Math.cos(phi1) * Math.cos(phi2) *
    Math.sin(deltaLambda/2) * Math.sin(deltaLambda/2);
    double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
    double distance = 6378137 * c; // Multiply with radius of the earth in meters
    return distance;
}

public static double calculateLightLatency(double lat1, double lon1, double lat2,
    double lon2) {
    double lightSpeed = 299792458.0;// Next: Compute light latency in milliseconds
    return calculateSphericalDistance(lat1, lon1, lat2, lon2) / lightSpeed * 1000;
}
```

Figure 5.5: Computing the light latency over the minimal distance on the orthodome between two locations using the haversine formula.

intriguing behavior, e.g., by reporting better latencies (or even 0 ms) for each other. The impact of this scenario is somewhat limited through the lower bound rule which raises false (unrealistically fast) reported latencies to the lower bounds of these latencies assuming the fastest possible internet speed based on the knowledge AWARE has on the locations of replicas.

5.3 Self-Optimization

The self-optimization of AWARE relies on performing *actions* to (1) adopt a new weight configuration and (2) select a new leader. For this purpose, replicas may periodically undergo deterministic reconfiguration, in which they synchronize and update the view. But before this happens, replicas must first analyze what the optimal configuration is and decide if the system should adopt it.

5.3.1 Optimizations

In general, AWARE incorporates two dynamic optimizations (see Figure 5.3):

- *Voting weights tuning*: By adjusting the voting weights, AWARE aims to enhance the latency experienced by clients across all sites. Previous research demonstrated that weighted replication can lower latency in WANs with a static configuration (Sousa and Bessani 2015). The goal of our research is to make this approach dynamic, i.e., to find an optimal weight distribution that minimizes the consensus latency of the system at runtime.

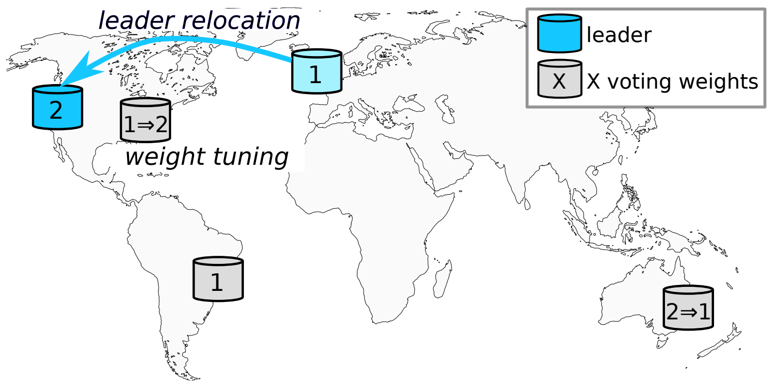


Figure 5.6: Optimizing a WHEAT configuration at run-time.

- *Leader relocation*: Relocating the leader to a well-connected site within the system has been shown to reduce the latency perceived by clients (Sousa and Bessani 2015; S. Liu and Vukolić 2017). AWARE can dynamically switch the leader's position if necessary.

Stability of the system. To ensure system stability we ideally want to prevent the system from jumping between system configurations that display almost equal performance. The overhead induced by letting replicas perform a reconfiguration (which demands a BFT-SMaRt *synchronization phase*) should be avoided in this scenario. Thus, an *optimization goal* α defines the threshold (relative to the ongoing system configuration) by which the predicted consensus latency needs to improve over the current consensus latency to justify a system reconfiguration.

The optimizations in AWARE are applied in logical *periods* of consensus instances. Reconfigurations are evaluated periodically, specifically once in each period. While the frequency of optimizations can be configured in our implementation, we use a default value similar to the checkpointing interval (for instance, say every 1000 consensus rounds). As a consequence, AWARE maintains stability by avoiding excessively frequent reconfigurations.

Leader selection. We employ the concept of *leader selection constraints*. In the AWARE system, we use an abstraction where the BFT SMR protocol provides an interface for selecting a leader. Specifically, we allow the protocol to present a set of potential leaders $\mathfrak{L}$, referred to as *leader candidates*, from which AWARE chooses one. This allows implementing orthogonal techniques to avoid "bad leaders" that can, e.g., be blacklisted for some time interval.

Algorithm 4: The utility *formQV* computes the times replicas form weighted quorums of $Q_v = 2t \cdot v_{max} + 1$ voting weights.

```
Data: replica set $\mathcal{R}$, latency matrix $\hat{M}^W$, times $(T_i^{current})_{i \in I}$ and voting weights $(v_i)_{i \in \mathcal{R}}$
Result: times $(T_i^{next})_{i \in \mathcal{R}}$ replicas can advance to the next protocol stage
for $i \in \mathcal{R}$ do                                                              1
    $received_i \leftarrow$ new PriorityQueue()                                          2
    for $j \in \mathcal{R}$ do                                                          3
        $received_i.add(\langle T_j^{current} + \hat{M}^W[j,i], v_j \rangle)$          4
for $i \in \mathcal{R}$ do                                                              5
    $weight \leftarrow 0$                                                                6
    while $weight < Q_v$ do                                                             7
        $\langle T_{next}, v_{next} \rangle \leftarrow received_i.dequeue()$            8
        $weight \leftarrow weight + v_{next}$                                          9
        $T_i^{next} \leftarrow T_{next}$                                               10
return $(T_i^{next})_{i \in \mathcal{R}}$                                              11
```

5.3.2 Consensus Latency Prediction

AWARE uses a model of the protocol to predict the expected latency of a consensus execution given the observed replica-to-replica latencies and a system configuration. In particular, the prediction model uses $\hat{M}^P$ and $\hat{M}^W$ to assess the consensus latency of different system configurations and then AWARE chooses a configuration that minimizes this latency. A fast configuration is one that *displays a low consensus latency from the perspective of the leader.* The leader can prepare and propose the next batch (Sousa and Bessani 2012; Bessani, Sousa, et al. 2014) as soon as the current consensus finishes, while, in the meantime, poorer connected replicas may still collect messages to form quorums. For the leader to make progress, it must also meet its quorum requirement of Q_v weights, which necessitates being well connected to replicas with high weights.

Time needed to collect a weighted quorum. Algorithm 4 is a utility that calculates, for each replica, the time at which it can advance to the next voting stage of the protocol by forming a weighted quorum consisting of $Q_v = 2tv_{max} + 1$ voting weights. This computation relies on the replica's current stage start time $T^{current}_{i \in I}$ and the latency matrix $\hat{M}^W$. The algorithm resembles a discrete, event-based simulation: Each replica retrieves messages from its priority queue, where incoming messages are sorted based on their arrival time and then accumulates the voting weights of these messages until Q_v is reached. The arrival time of the last message necessary to reach this quorum determines when a replica can proceed to the next protocol stage as a result.

Consensus latency. Algorithm 5 simulates the consensus protocol execution to estimate the latency of the leader's consensus for a specific configuration. The algorithm begins by calculating the timestamps at which each replica receives the leader's proposal (lines 4-5). Subsequently, it employs Algorithm 4 to determine the time taken by replicas to complete two rounds of voting, the WRITE stage (line 6) and the ACCEPT stage (line 7), respectively.

Algorithm 5: *PredictLatency* computes the consensus latency (amortized over multiple rounds).

```
Data: replica set $\mathcal{R}$, leader $p$, system sizes $n$, $t$, $\Delta$, weight config. $W = \langle R_{max}, R_{min} \rangle$, latency
      matrices for PROPOSE $\hat{M}^P$ and WRITE $\hat{M}^W$, consensus rounds $r$
Result: consensus latency of the AWARE leader
$v_{max} \leftarrow 1 + \frac{\Delta}{t} \quad v_{min} \leftarrow 1 \quad v_i \leftarrow \begin{cases} v_{max}, & \text{if } i \in R_{max} \\ v_{min}, & \text{otherwise} \end{cases}$    1
$\forall i \in \mathcal{R}: \ offset_i \leftarrow 0$    2
while $r > 0$ do    3
    for $i \in \mathcal{R}$ do    4
        $T_i^{PROPOSED} \leftarrow max(\hat{M}^P[p, i], offset_i)$    5
    $(T_i^{WRITTEN})_{i \in \mathcal{R}} \leftarrow formQV(\mathcal{R}, \hat{M}^W, (T_i^{PROPOSED})_{i \in \mathcal{R}}, (v_i)_{i \in \mathcal{R}})$    6
    $(T_i^{ACCEPTED})_{i \in \mathcal{R}} \leftarrow formQV(\mathcal{R}, \hat{M}^W, (T_i^{WRITTEN})_{i \in \mathcal{R}}, (v_i)_{i \in \mathcal{R}})$    7
    for $i \in \mathcal{R}$ do    8
        $offset_i \leftarrow T_i^{ACCEPTED} - T_p^{ACCEPTED}$    9
    $consensusLatencies_r \leftarrow T_p^{ACCEPTED}$    10
    $r \leftarrow r - 1$    11
return average of $consensusLatencies$    12
```

Additionally, this algorithm can calculate the average leader consensus latency amortized across multiple rounds r, or single-shot consensus latency in the special case of $r = 1$. It should be noted that replicas reach consensus at different points in time (as illustrated in Figure 5.7). If the time difference between when the leader p makes a decision and when replica i makes a decision exceeds the propose latency $\hat{M}^P[p, i]$, then replica i may receive a proposal for the next consensus instance first but must wait for the completion of its previous consensus before broadcasting its

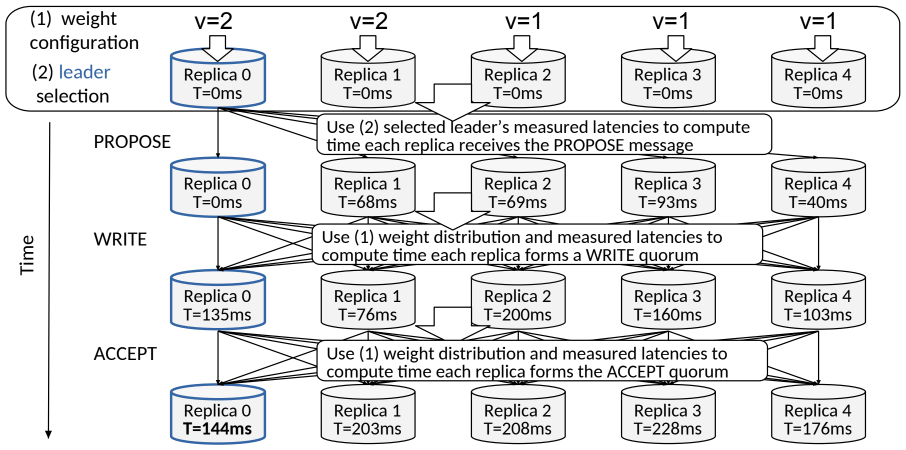


Figure 5.7: Computing the consensus latency of WHEAT for a given configuration.

WRITE message. This situation can potentially slow down the leader's pace, but only if replica i is included in the quorum to achieve Q_v voting weights, indicating that replica i's message serves as the determining vote for the speed of quorum formation. In our calculations for a series of rounds, we account for this by calculating the *offsets*. These represent the additional time needed by other replicas to complete their consensus relative to the leader.

On a side note, optimizing the system opens a wide configuration space to explore. An *AWARE configuration* defines the weight configuration and chooses a leader. The number of possible configurations is the number of weight configurations multiplied with the number of possible leaders (we only consider the v_{max}-replicas for leader nomination):

$$\binom{3t+1+\Delta}{2t} \cdot 2t = \frac{\prod_{i=2t}^{3t+1+\Delta} i}{(t+1+\Delta)!} \tag{5.7}$$

This number quickly grows with increasing system size. For instance, it gives just 20 configurations for a $n = 5, t = 1, \Delta = 1$ system but 504 configurations for a $n = 9, t = 2, \Delta = 2$ system.

Traversing the entire search space of configurations becomes impractical in systems with a large number of replicas. To encounter this problem, AWARE uses heuristics for approximating the optimum, e.g., simulated annealing using *PredictLatency* to assess the goodness of a found solution. We explain, implement, and validate this approach in Section 5.7.

5.4 The AWARE Protocol

AWARE uses only a few modifications towards WHEAT (Sousa and Bessani 2015) which are mostly realized in a distinct module `aware` that allows each replica to maintain and collect monitoring information as part of the replica state (which makes replicas aware of their environment) as well a prediction model that can be used to compute consensus protocol latency.

To ensure uniformity of measurement data across all replicas following a particular consensus instance, AWARE synchronizes the recorded latency information through total order broadcast. To guarantee determinism, AWARE establishes a deterministic function for predicting consensus latency, which is called to assess the quality of AWARE configurations (see Algorithm 5).

In a reconfiguration step, replicas modify the *view* object to update weights that are relevant for computing quorums. These reconfigurations take place only after performing calculations based on logical protocol rounds (consensus instances) rather than relying on time intervals. An overview of AWARE is presented in Figure 5.8 (for the sake of a simpler exposition, it hides a few details, e.g., about the collection of latency information that involves the sending of response messages).

Replica Behavior

Self-Monitoring:

1. Each replica $i \in \mathcal{R}$ collects its latency measurements (a moving median) for PROPOSE (L_i^P) and WRITE messages (L_i^W) in a vector, respectively:
$$L_i^P = \langle l_{i,0}^P, ..., l_{i,n-1}^P \rangle$$
$$L_i^W = \langle l_{i,0}^W, ..., l_{i,n-1}^W \rangle$$
2. Periodically, each replica $i \in \mathcal{R}$ disseminates its vectors L_i^P and L_i^W with total order by calling *invokeOrdered*$(\langle \text{MEASURE}, L_i^P, L_i^W \rangle)$;
3. Once some replica $i \in \mathcal{R}$ decides a batch, that batch may contain some messages $\langle \text{MEASURE}, L_j^P, L_j^W \rangle$ from some replica $j \in \mathcal{R}$. Then, i uses these vectors to update its synchronized matrices M_i^P and M_i^W, i.e., for replicas $k = 0, .., n-1$ assigning
$$M_i^W[j,k] = L_j^W[k]$$
that is the information that i now has about the latency between replica j and all other replicas measured by j. This applies also for the latency information for PROPOSE (M_i^P).
4. Every c of consensus instances, all replicas have the same matrices M^P and M^W, e.g., with $M^W[i,j] = L_i^W[j]$ if replica i sent its WRITE measurements within the last c consensus instances, or $M^W[i,j] = +\infty$ if i did not. The same applies to M^P. These matrices are sanitized to avoid the influence of malicious replicas (see §5.2.2), yielding $\hat{M}^P$ and $\hat{M}^W$.

Self-Optimization:

1. After every c consensus instances, every replica solves the following optimization problem, where *PredictLatency* (Algorithm 5) is a deterministic function for predicting the latency of the consensus protocol using the latencies in $\hat{M}^P$ and $\hat{M}^W$, as well as a set of weight distributions $W \in \mathfrak{W}$ and a set of leader candidates $l \in \mathfrak{L}$:
$$\langle \hat{l}, \hat{W} \rangle = \underset{W \in \mathfrak{W}, l \in \mathfrak{L}}{\arg\min} \; \mathit{PredictLatency}(l, W, \hat{M}^P, \hat{M}^W) \tag{5.8}$$
2. The configuration $\langle \hat{l}, \hat{W} \rangle$ with optimal leader consensus latency is the one selected for the next reconfiguration if the predicted latency is better than the current configuration by a threshold α (optimization goal). Since this procedure is deterministic, the $\langle \hat{l}, \hat{W} \rangle$ is the same in all replicas and the *action* of initiating a reconfiguration is deterministic too.
3. In case the replicas find a faster weight configuration, they update their view to respect the new voting weights for the following consensus instances. Optionally, if the system uses leader relocation, the replicas trigger a synchronization phase to elect a faster leader.

Figure 5.8: An overview of AWARE.

5.5 Implementation in WHEAT

We implemented a prototype of AWARE on top of WHEAT, an extension of BFT-SMaRt that incorporates a weighted quorum scheme (Sousa and Bessani 2015). To maintain the modularity of BFT-SMaRt most of the changes were realized in an specific module `aware`, which consists of the following submodules:

- `aware.decisions`: This submodule includes code for assessing the performance of system configurations (i.e., the *prediction model*), as well as the code used for periodically attempting optimizations and initiating system reconfigurations.
- `aware.monitoring`: This submodule includes a *measurements* instance that continuously collects measurements (the local view of measurements collected by an individual replica), a *monitor* object, that stores and maintains the global view (the latency matrices), as well as methods for the sanitization of measurements, and a *synchronizer* that disseminates collected

measurement data with total order.

- `aware.messages`: This submodule defines the new messages that are introduced with AWARE used for monitoring the consensus protocol.

AWARE is essentially a variant of WHEAT that supports the dynamic assignment of weights and leader placement and employs latency measurements as a decision-making basis. Most of the added code relates to the assessment of system configurations (for which a simulation-based prediction model is implemented) and handling the optimizations as well as implementing the monitoring techniques described in Section 5.2.

Note that, to obtain a correct implementation, there are a few cross-cutting modifications necessary in some of the existing subprotocols of BFT-SMaRt. For instance, *state transfer* needs to transfer the monitoring data as well to ensure recovering replicas that fetch the state from others can make consistent decisions, for which they need identical latency matrices. Since replicas use proofs to validate consensus decisions, they need to remember used views (and for which consensuses they were used) to ensure they remember a weight assignment that was potentially used in the past to constitute a quorum.

Just like in BFT-SMaRt, the AWARE implementation employs TLS to secure all communication channels as well as the elliptic curve digital signature algorithm (ECDSA) for signatures and SHA256 for hashes.

5.6 Evaluation

In this section, we evaluate our approach through a series of experiments that we conduct on a WAN that employs multiple AWS regions. Firstly, we quantify the performance of different WHEAT configurations to find out how much latency reduction can be achieved by optimizing the system configuration. Secondly, we assess the accuracy of our implemented prediction model by comparing predictions of consensus latency with measurements taken from real protocol runs. Thirdly, we investigate the effect that optimization has on the request latency experienced by clients that are spread around the world. Fourthly, we induce events (i.e., a replica failure) into a running AWARE system to evaluate its runtime behavior during periodic self-optimizations. Fifthly, we measure throughput and examine the monitoring overhead within the system. Finally, we reason about the system's behavior in failure scenarios.

Setup. By default, we use the Amazon AWS cloud to deploy our EC2 instances in specific regions. We choose the following numbered regions for our setup (this concrete deployment is also shown in Figure 5.6):

(0) *Oregon*

(1) *Ireland*

(2) *Sydney*

(3) *São Paulo*

(4) *Virginia*

In each of these regions, we deploy a virtual machine (VM) to construct our globally distributed replicated system. Each VM serves the dual purpose of hosting a replica and acting as a client for conducting latency measurements. For our latency experiments, we do not require high-end hardware, so we opt for the *t2.micro* instance type. These instances come with 1 vCPU and 1 GB of RAM. We use a configuration where $t = 1$ and $\Delta = 1$ thus mimicking the evaluation setup of WHEAT (Sousa and Bessani 2015).

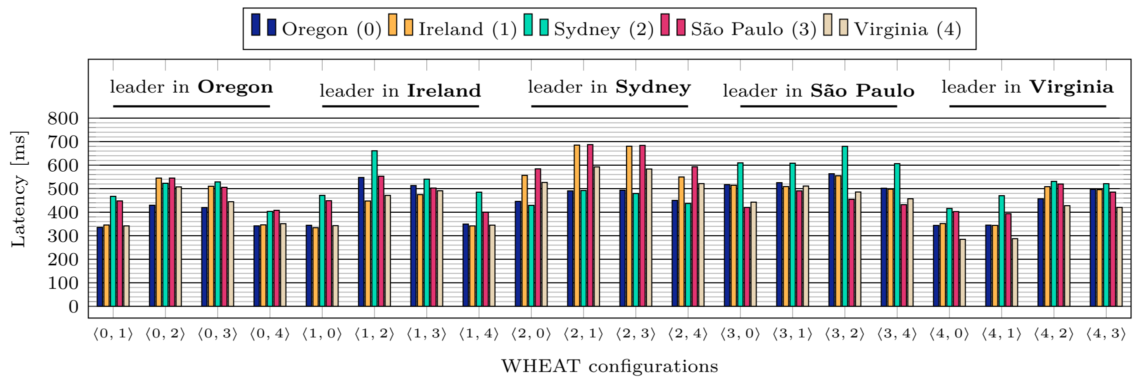


Figure 5.9: Clients located in different regions (Oregon, Ireland, Sydney, São Paulo and Virginia) report their observed **average** request latency from 11th to 90th percentile.

Method. When it comes to latency measurements, we are particularly interested in two metrics:

- *Consensus latency* defines the time between a leader starting consensus (by initiating the broadcast of a proposal) and deciding the consensus.
- *Request latency* is the time between a client submitting an operation to the service and accepting the result of that operation (this requires collecting replies from several replicas).

The replicas calculate the average consensus latency based on a sample of 1000 consensus instances. Meanwhile, clients initiate at least 1000 requests simultaneously and continue sending requests until each client completes its measurements. If a client request reaches the leader replica while a consensus instance is currently in progress, it may experience a delay before being included in a batch. We employ synchronous clients that block (meaning wait) for the response before sending the next request. These clients also wait for a random time delay between 0 and 150 ms before sending each subsequent request. Additionally, to mitigate the impact of outliers, the clients calculate the average latency based on the 11th to 90th percentile of perceived request latencies.

5.6.1 Is a Dynamic Approach even Necessary?

First, we validate the motivation behind a dynamic approach: "*Is the performance variation between the individual system configurations large enough to justify a dynamic optimization approach?*". To justify a dynamic approach in autonomously discovering a high-performing configuration, we begin by highlighting the disparity among all possible WHEAT configurations for our $n = 5$ setup.

Results. Figure 5.9 provides a visual representation of the observed latencies experienced by clients across different regions. Each configuration is denoted by a tuple $\langle l, r\rangle$, where l represents the leader, and both replicas l and r are selected to have maximum weights, i.e., of $v_{max} = 2$. Each number corresponds to a region as explained in the setup.

A clear distinction can be observed among the configurations. The optimal configuration, represented as $\langle 4, 0\rangle$, exhibits an average latency of 360 ms across all clients. In comparison, the configuration with the median performance, denoted as $\langle 3, 4\rangle$, operates at a latency of 499 ms, while the least favorable configuration, $\langle 2, 1\rangle$, necessitates 590 ms.

The best WHEAT configuration, $\langle 4, 0\rangle$, achieves a latency improvement of 38.7% compared to the median configuration and a 63.9% improvement over the worst configuration.

In Figure 5.9, we can make the following observations:

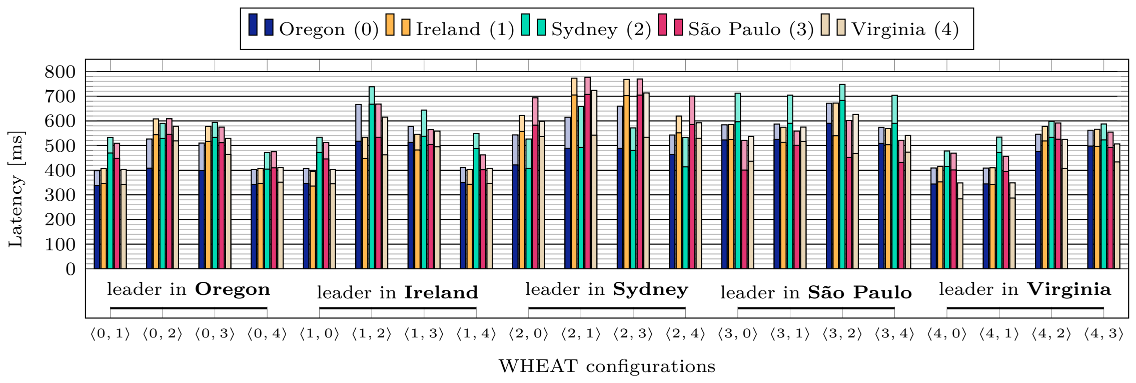


Figure 5.10: Clients located in different regions (Oregon, Ireland, Sydney, São Paulo and Virginia) report their observed **median** (and **90th** percentile in light color) request latency.

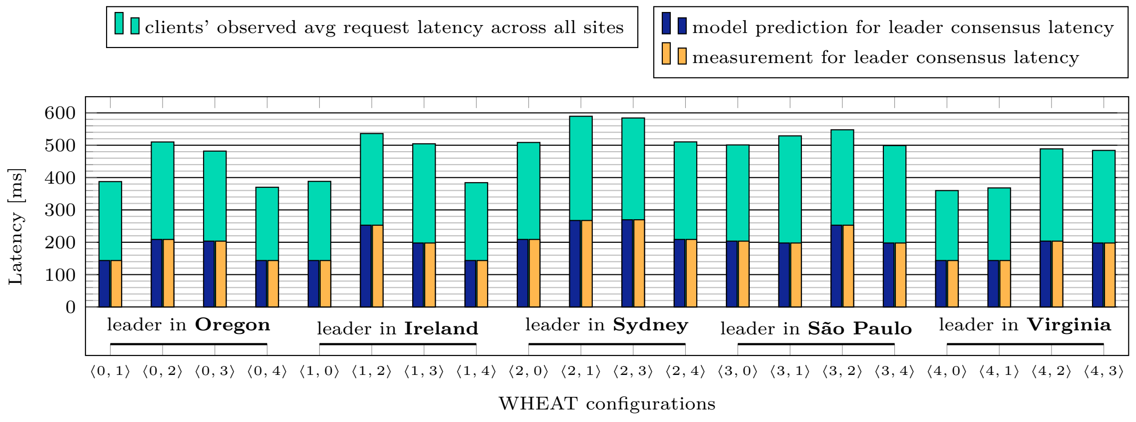


Figure 5.11: Comparison between predicted consensus latency, measured consensus latency, and clients' request latency.

1. *Latency reduction through voting weight tuning:* Modifying the weights shows promise as an optimization technique for reducing latency, even when the leader remains fixed. This becomes evident when comparing different configurations with the same leader.

2. *Optimal latency may require leader selection:* Leaders located in Sydney or São Paulo do not possess sufficiently fast connectivity with the rest of the system. Relocating the leader seems to be necessary to improve latency for all clients.

3. *Client-to-leader co-locality:* Clients situated in close proximity to the leader generally experience lower request latencies compared to other clients within a specific configuration. However, co-locating the leader with some specific client does not guarantee that this specific client observes the lowest possible latency compared to other configurations where there is no co-locality. For instance, a client in Sydney records a latency of 492 ms in configuration $\langle 2, 1\rangle$ (with the leader being co-located), while it achieves its best performance (among all configurations) in $\langle 0, 4\rangle$ with a latency of 403 ms (where *Oregon* is the leader).

4. *Few fast configurations dominate over poorer configurations:* As we can see in Figure 5.10 a few fast system configurations outperform a larger number of less favorable configurations. These configurations are the ones that AWARE estimates as being fast WHEAT configurations in regard to *consensus latency* and selects for optimization.

Figure 5.10 details the averaged latency observations made by Figure 5.9 by displaying both the median and 90th percentile of observed request latencies in each client.

5.6.2 Accuracy of Consensus Latency Prediction

The objective of our approach is to identify a configuration that minimizes the consensus latency of the leader. By using our prediction model (Algorithm 5), we calculate these latencies for all configurations.

Results. In our analysis, we compare the predictions generated by our model with the consensus latency measured by the respective leader for each configuration conducted in our experiment (refer to Figure 5.11). Across these configurations, our average prediction error amounts to only 1.08% Among the configurations, the highest prediction error occurs for $\langle 4, 0\rangle$ at 3.22%. Considering that our predictions are based on latency measurements that exhibit smaller variations, we contend that these results are reasonable for selecting a fast system configuration.

It is important to note that while AWARE may not always select the absolute best configuration, it tends to choose a configuration that is in close proximity to the optimum. In our given scenario, AWARE has the option to select any configuration among $\langle 0, 1\rangle$, $\langle 0, 4\rangle$, $\langle 1, 0\rangle$, $\langle 1, 4\rangle$, $\langle 4, 0\rangle$, or $\langle 4, 1\rangle$, based on its prediction of a leader consensus latency of 143.5 ms amortized over 1000 consensus instances.

During our experiment, the measured latencies for these "optimal candidates" range from 141 ms (in the case of $\langle 1, 0\rangle$) to 148 ms (in the case of $\langle 4, 0\rangle$). If there exists an optimal configuration that includes the current leader, AWARE prefers selecting it rather than configurations that necessitate a leader change. As an additional note, the median predicted leader's consensus latency is 202 ms for $\langle 3, 4\rangle$, while the worst predicted latency is 271 ms for $\langle 2, 1\rangle$.

5.6.3 Clients' Observed Request Latency

Figure 5.11 presents a comparison of the observed request latency (averaged across all sites) for different configurations, along with the corresponding model predictions and measurements for consensus latency. As expected, the speed of achieving consensus significantly contributes to the total latency. Notably, we observe a strong positive correlation, denoted as

$$\rho(L^{MP}, L^{CR}) = \frac{Cov(L^{MP}, L^{CR})}{\sqrt{Var(L^{MP}) \cdot Var(L^{CR})}} = 0.961 \tag{5.9}$$

between our series of model predictions for leader consensus latency, L^{MP}, and the series of measurements for average clients' request latency, L^{CR}. This correlation indicates that *faster consensus is beneficial for geographically distributed clients.*

5.6.4 Runtime Behavior

In our standard environment, we deploy AWARE and closely monitor its behavior throughout the lifespan of the system while we induce *events*. Throughout this period, we observed significant variations in the request latencies experienced by the clients. These variations primarily stem from the waiting time that occurs when a request reaches the leader. Since all clients send requests simultaneously and the leader batches them, a client's request may need to wait at the leader until the ongoing consensus is completed before being processed in the subsequent consensus. The duration of this waiting time fluctuates and depends on the timing of the request in relation to the start of the next consensus. We induce the following events to evaluate AWARE's automated reactions (which are marked blue (ⓧ) in Figure 5.12) to specific conditions:

① Action: We deploy AWARE in a slow system configuration $\langle 2, 3\rangle$ with the leader being *Sydney* and *São Paulo* is the other replica (besides the leader) to have maximum voting weights.

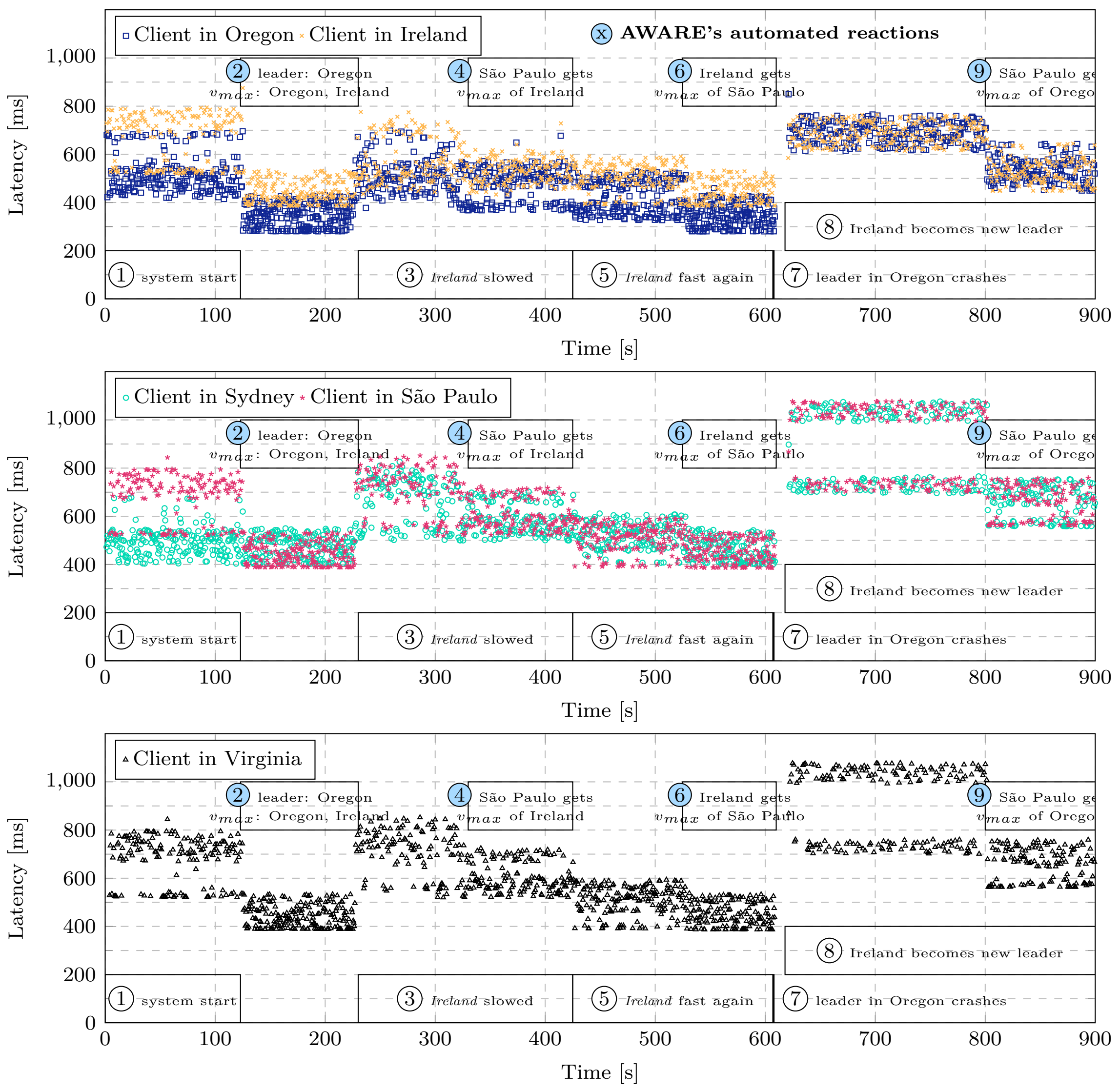


Figure 5.12: Induction of events into a running AWARE system.

② Reaction: After a first round of measurements, the first optimization point is reached (after $c = 500$ consensus instances), AWARE determines that *Oregon* and *Ireland* are more optimal to have the high weights and thus switches its configuration to $\langle 0, 1 \rangle$, which leads to *latency reductions experienced by clients across all system sites.*

③ Action: We induce a network disturbance, i.e., we apply an outgoing delay of 120 ms and 20 ms jitter to the replica in *Ireland*, thus making its network links slower:

```
tc qdisc replace dev eth0 root netem delay 100ms 20ms distribution normal
```

(the client and replica of Ireland are not co-located on the same VM and the client is not affected by this).

④ Reaction: AWARE re-distributes one of the v_{max} to *São Paulo* while *Ireland's* weight becomes v_{min}. Clients experience a small improvement in their request latencies.

⑤ Action: We undo the network disturbance for *Ireland*, hence the network stabilizes and communication links of the replica in *Ireland* are now as fast as they were initially.

⑥ Reaction: The monitoring procedure of AWARE spots this improvement and re-distributes the v_{max} of *São Paulo* back to *Ireland*, as it predicts a latency reduction for this configuration.

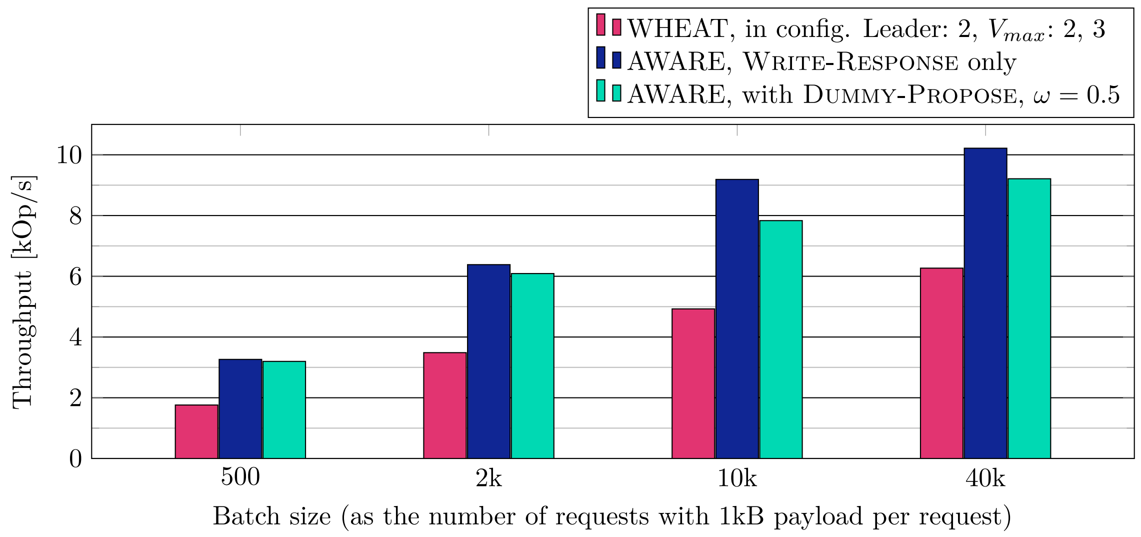


Figure 5.13: Maximum throughput comparison.

After the reconfiguration, clients experience faster request latencies identical to the situation after the first reconfiguration (see Event (2)).

(7) Action: We crash the leader *Oregon* (which has v_{max}).

(8) Reaction: After a few seconds, the request timers of replicas expire and then BFT-SMaRt triggers the leader change protocol: *Ireland* becomes the next leader. Since $t \cdot v_{max}$ voting weights become unavailable, all *remaining correct replicas need to access the same quorum Q_v* (consisting of all 3 v_{min} replicas and the leader). This leads to clients experiencing higher request latencies.

(9) Reaction: AWARE assigns the v_{max} of the crashed replica to a former v_{min} replica, *São Paulo*, hence *restoring some degree of variability in quorum formation.* Replicas now can form smaller quorums. Subsequently, clients across all regions experience latency improvements.

5.6.5 System Throughput

In this experiment, we evaluate the impact of system reconfigurations toward the maximal achievable throughput of AWARE. Fur this purpose, employ AWS instances of the type *c5.xlarge* which come with 4 vCPU and 8 GB of RAM. We use the same regions to place in each region a VM for the client a dedicated VM for the replica. On the client VMs we systematically increase the number of clients until we reach the saturation point of the system, at which the observed throughput does not increase anymore. The throughput is measured by the leader replica. In this experiment, we use asynchronous (i.e., non-blocking) clients which submit requests with a payload of size 1 kB after randomly waiting between 0 ms and 10 ms. The server responses also carry 1 kB of payload.

We conducted a comparison among three different variations:

1. WHEAT in a suboptimal system configuration ($\langle 2, 3\rangle$),
2. AWARE using the WRITE-RESPONSE monitoring approach and *after* reconfiguration to $\langle 0, 4\rangle$,
3. AWARE configured with $\langle 0, 4\rangle$ and the DUMMY-PROPOSE feature enabled. We limit monitoring overhead to $\omega = 0.5$. In this configuration every second consensus instance a replica determined by a rotation scheme broadcasts a DUMMY-PROPOSE alongside the leader;

In theory, faster consensus instances and increasing the batch size (the number of decided requests per consensus) can both enhance system throughput. However, larger batch sizes also result in higher latencies for PROPOSE operations due to bandwidth constraints, which can in turn increase consensus latency. Our experiments demonstrate (see Figure 5.13) that (1) reducing consensus latency indeed has a positive impact on throughput across various batch sizes, and (2) enabling the DUMMY-PROPOSE feature introduces some monitoring overhead which is noticeable, but still passable, given that AWARE is proposed as a latency optimization technique[3].

5.6.6 Impact of Faulty Replicas

AWARE tolerates up to t Byzantine faults. If replicas crash in WHEAT, then the weights they possess become temporarily unavailable for quorum formation within the system until the crashed replicas are repaired and re-integrated into the system. Interestingly, AWARE can detect crash failures through its self-monitoring procedure and swiftly re-distribute voting weights (imagine a crashed replica with v_{max} weight, it can be given to a former v_{min}-replica) to mitigate performance degradation. But things are different in a malicious setting. Byzantine replicas might try to prevent AWARE from redistributing weights. In the following, we discuss both scenarios.

Crash Faults. In the first scenario, AWARE identifies unavailability by recognizing that crashed replicas neither communicate their measurements nor acknowledge protocol messages. Consequently, AWARE's prediction model will model a crashed replica with an ∞-value. Assume the crashed replica held a v_{max} weight, AWARE reallocates this weight to a more efficient replica (as we can see in Figure 5.12, Event 9), thereby reintroducing some degree of flexibility in quorum formations. If a previously fast replica that crashed is subsequently repaired and reintegrated into the system, then it might regain its original voting weight. In summary, a crash fault only leads to $v_{min} = 1$ weight becoming (temporarily) inaccessible for quorum formation while it is a weight of either v_{min} or v_{max} in WHEAT because WHEAT does not allow to redistribute weights at runtime.

Malicious Faults. In a malicious setting, Byzantine replicas may purposely stay silent and not participate in consensus (i.e., not broadcasting WRITE messages), yet still transmit response messages to other replicas and publish latency vectors that would attribute themselves beneficially low latencies. For instance, if $t > 1$, pairs of malicious replicas might claim that their inter-replica communication latencies are nearly 0. The sanitization procedure of AWARE's monitoring strategy can detect and reject the lying behavior of Byzantine replicas to a limited extent by employing their locations to estimate their best possible network link latency as a lower bound. In particular, Byzantine replicas can not pretend that their network links to each other are faster than physically possible (links can not transmit faster than the the speed of light, whereby $\frac{2}{3}$ of lightspeed is accepted as the upper bound on data transmission speed for the internet (Cangialosi et al. 2015; Kohls and Diaz 2022)).

AWARE's capability to promptly detect the unavailability of malicious replicas becomes hindered, thereby constraining its ability to exclusively redistribute voting weights among correct replicas. In a worst-case scenario assumption, a total of t malicious replicas might collectively control up to $t \cdot v_{max}$ voting weights, compelling correct replicas to use the sole remaining fallback quorum, composed of $t \cdot v_{max} + (t+1+\Delta)v_{min}$ voting weights contributed by all $2t+1+\Delta$ correct replicas, to advance.

Nonetheless, we can ensure the system's availability. Note that failures generally limit the variability in quorum formation across all consensus protocols that employ quorum systems. This holds irrespective of whether these protocols use egalitarian or weighted quorums. Yet from a performance standpoint, we might at least want to make sure that the optimized system does not perform worse than its non-optimized counterpart. We tackled this challenge in a future work (which we

[3]The DUMMY-PROPOSE can help to improve the accuracy of finding the best leader position. It only leads to an improvement compared to not using it in rare situations where the PROPOSE latency can not be reasonably well estimated through the collected latencies measured by using the WRITE-RESPONSE.

n	8	9	10	11	12	13	14	15	16	17
t	2	2	2	3	3	3	4	4	4	4
Δ	1	2	3	1	2	3	1	2	3	4

Table 5.1: Exemplary system sizes.

introduce in Chapter 6) by introducing *consensus latency expectations* which allows us to detect if the system is slower than it should be (a similar idea was originally proposed by RBFT (Aublin, Mokhtar, et al. 2013)) and in that case, reconfigure the system.

5.7 AWARE Scalability Enhancements

In the previous section, we studied the AWARE system for relatively small system setups with $n = 5$ replicas only. In this section, we first want to extend our ambitions towards larger systems. First, we explore which enhancements are needed to use the AWARE methodology in larger systems where the fast-growing configuration space complicates finding a good configuration. Second, we carry out experiments involving numerous geographic regions on the AWS infrastructure. These experiments aim to demonstrate that employing adaptive weighted replication is an effective strategy for achieving geographically scalable BFT consensus.

As mentioned previously, we showed the rapid growth of the configuration space due to various weight distributions being possible. The vast configuration space makes an exhaustive search impractical because, for every configuration, Algorithm 5 needs to be computed. This is due to the combinational nature of these configurations, which involve choosing $2t$ replicas to assign a weight of v_{max} from a combined pool of $3t + 1 + \Delta$ total replicas within the system. The exponentially increasing number of configurations for exemplary system sizes, as presented in Table 5.1, is visually represented in Figure 5.14a.

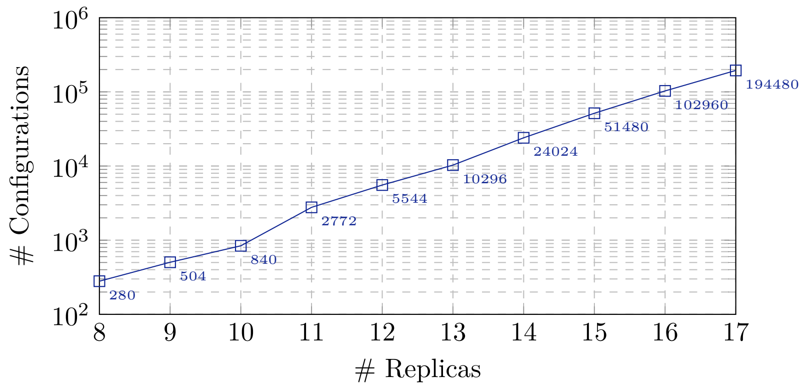
5.7.1 Simulated Annealing

To tackle this issue, we employ a heuristic method to efficiently navigate the exploration space, called *simulated annealing* (Kirkpatrick et al. 1983). Simulated annealing (SA) is a generic method that can approximate the global optimum of a specific function (outlined in Algorithm 6).

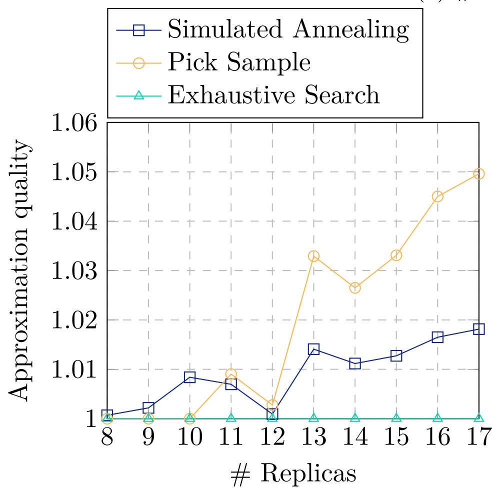
Primarily, SA works similar to a local search strategy aimed at iteratively enhancing a *current solution* x by making slight random modifications, referred to as *neighbor picking*, resulting in a new configuration denoted as x'. If this new configuration x' improves the function's output (in our case, it results in a lower predicted latency), the simulated annealing algorithm adopts x' for further exploration. Yet, if the function output worsens due to x', the algorithm may still utilize it for exploration with a probability of acceptance. This acceptance probability depends on the existing *temperature* τ, a value that steadily decreases, and the energy of the solution – essentially, the difference between the predicted latency of x' and x. By allowing temporary worsening, simulated annealing can make *jumps*, enabling it to move away from local optima and possibly discover the global optimum. Nevertheless, as the temperature decreases, such jumps occur less frequently.

Termination criteria for this search methodology can be set, for example, by employing a temperature threshold. This ensures that termination is guaranteed since the temperature declines at a fixed rate. In the simulated annealing heuristic, we employ the following parameterization:

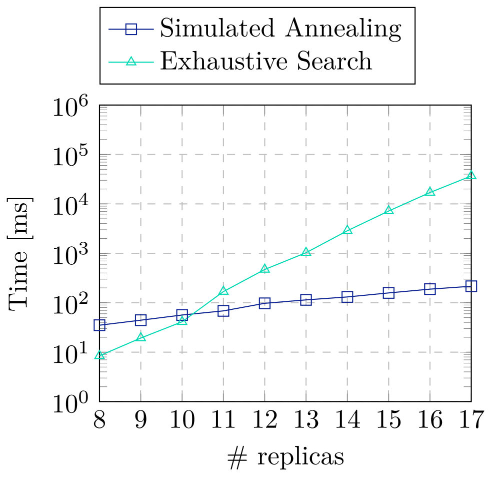
- start temperature $\tau = 120$
- cooling rate $\theta = 0.0055$
- end temperature (threshold) $\bar{\tau} = 0.2$



(a) # Configurations.



(b) Approximation quality.



(c) Computation time.

Figure 5.14: Heuristics can help efficiently traverse the configuration space to find a good solution.

We selected these values through empirical methods. Using these parameters, the algorithm concludes its exploration after investigating 1962 system configurations.

Given that simulated annealing operates probabilistically, it necessitates generating a sequence of random numbers throughout its execution. However, to ensure that replicas consistently arrive at a consistent solution, a pseudo-random number generator (*PRNG(s)*) must be used, producing the same sequence of random numbers across all replicas when provided with the same input seed s. In our case, we let replicas use the consensus ID (which replicas determine for potential reconfigurations during specific consensus instances) as the seed for generating these numbers.

5.7.2 Approximation Quality

We proceed to empirically validate the goodness of solutions found by our heuristic method denoted as *Simulated Annealing* (SA) using the previously described parametrization. This validation is accomplished through a comparative analysis of the predicted consensus latency associated with the approximate solutions derived via SA in contrast to the optimal solutions attained using an *Exhaustive Search* (ES) technique. Furthermore, the comparison is extended to encompass a third strategy, labeled as *PickSample*, which systematically navigates the configuration space with uniform increments, probing the equivalent number of configurations as SA.

Our evaluation computes the average deviation from the optimal solution provided by ES through conducting simulations on a pool of 1000 randomly generated system environment (latency maps), created for each system size n. The findings, depicted in Figure 5.14b, illustrate that, on average,

Algorithm 6: *SimulatedAnnealing* is a heuristic to efficiently traverse the search space of all system configurations $Confs$ and find an optimized configuration.

```
Data: replica set $\mathcal{R}$, system sizes $n$, $t$, $\Delta$, latency matrices for PROPOSE $\hat{M}^P$ and WRITE
      $\hat{M}^W$, consensus id $c$, start temperature $\tau$, cooling rate $\theta$, end temperature threshold $\bar{\tau}$
Result: best (approx.) performing configuration found
// Initially, replicas deterministically start the search with the same,              1
   current config.
$x \leftarrow$ current config $x_0 \in Confs$ ;                                             2
$x.prediction \leftarrow predictLatency(\mathcal{R}, x, \hat{M}^P, \hat{M}^W, n, t, \Delta)$ ;            3
$x_{approx} \leftarrow x$ ;                                                                 4
$random \leftarrow$ new Random(c) ;                                                         5
while $\tau > \bar{\tau}$ do                                                                6
    // Create a ramdom neighbor configuration by shifting a weight $v_{max}$ to a      7
       $v_{min}$ replica
    $replicaFrom \leftarrow x.R_{max}[random.nextInt(2t)]$ ;                                 8
    $replicaTo \leftarrow x.R_{min}[random.nextInt(n-2t)]$ ;                                9
    $x' \leftarrow x.swap(replicaFrom, replicaTo)$ ;                                        10
    if $replicaFrom$ is leader then                                                        11
        $x'.setLeader(replicaTo)$ ;                                                         12
    $x'.prediction \leftarrow predictLatency(\mathcal{R}, x', \hat{M}^P, \hat{M}^W, n, t, \Delta);$ ;       13
    // If new solution is better, accept it                                              14
    if $x'.prediction < x.prediction$ then                                                  15
        $x \leftarrow x'$ ;                                                                 16
    else                                                                                    17
        // Compute an acceptance probability                                             18
        $\clubsuit \leftarrow random.nextDouble()$ ;                                        19
        if $exp(\frac{-(x'.prediction - x.prediction)}{\tau}) > \clubsuit$ then              20
            $x \leftarrow x'$ ;                                                             21
    if $x'.prediction < x_{approx}.prediction$ then                                         22
        $x_{approx} \leftarrow x'$ ;                                                        23
    // Cool down the system                                                              24
    $\tau \leftarrow \tau \cdot (1-\theta)$ ;                                               25
return $x_{approx}$ ;                                                                       26
```

the predicted consensus latencies of solutions produced by SA are only slightly higher than the optimal latencies found with ES, within a margin of less than 1.02. Notably, SA outperforms the simplistic sampling approach, as the solutions obtained through *PickSample* exhibit a larger deviation from the optimal solutions, reaching up to an approximation of only 1.05.

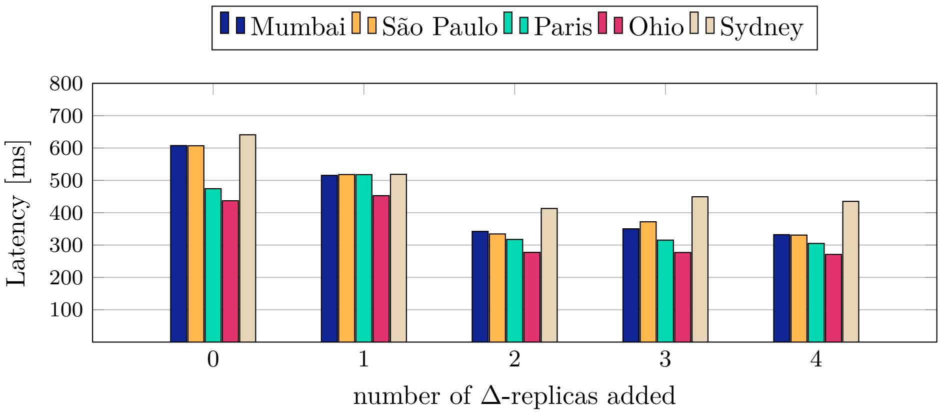
5.7.3 Computation Time

Furthermore, we assess the average time required for a replica to find a solution while employing an Intel i7-4790 processor with 3.60 GHz. This evaluation involves the utilization of 1000 randomly generated environments for each system size n. We compare the two strategies, SA and ES in Figure 5.14c.

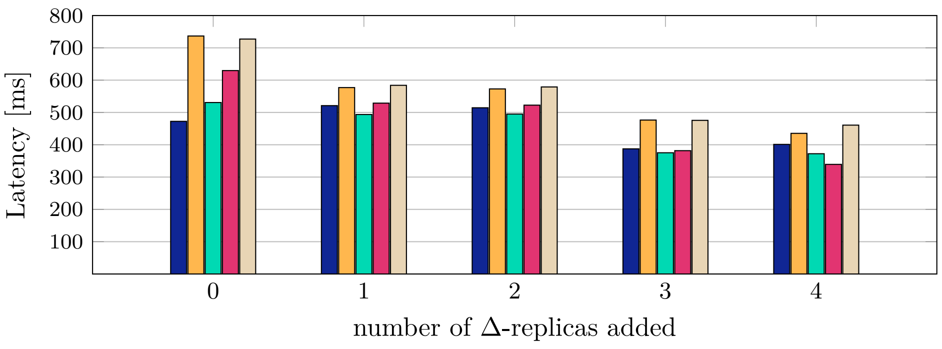
As anticipated, the time required by ES experiences exponential growth as n increases. On the other hand, SA only traverses a consistent number of configurations; nevertheless, the time increases due to the increased computation time needed by the *PredictLatency* function for predicting the latency of larger systems (with more replicas). The prediction function essentially simulates the execution of the consensus protocol, resulting in a time complexity of $\mathcal{O}(n^2 log(n))$. This complexity arises from the use of a priority queue of messages maintained by each replica, ordered by ascending arrival time (a min-heap), which necessitates $\mathcal{O}(nlog(n))$ time to construct and we consider each of the n

t	**Initial replica set**	**Δ-replicas**
2	Mumbai, São Paulo, Paris, Ohio, Sydney, Tokyo, Canada	Virginia, Ireland, Stockholm, Frankfurt
3	Mumbai, São Paulo, Paris, Ohio, Sydney, Tokyo, Canada, Bahrain, Stockholm, Seoul	Frankfurt, Virginia, Ireland, Oregon

(a) Test environments.



(b) $t = 2$ configuration.



(c) $t = 3$ configuration.

Figure 5.15: Latency results of AWARE experimentally tested in larger-scale setups.

replicas during the simulation to compute the times at which they can collect their quorums.

Important to note is, that the number of configurations that SA probes depends on the exact parameterization of SA, i.e., start temperature, end temperature, and the cooling rate. An adjusted parameterization that probes more configuration can make sense to improve the approximation quality at the cost of a higher computation time. In total, the time complexity of our SA-based prediction model is in $\mathcal{O}(n^2 log(n))$.

5.7.4 Experiments on AWS

Within this section, we present a series of experiments conducted within the AWS cloud infrastructure. The objective of these experiments is to scrutinize the behavior of adaptive weighted replication in scenarios where the replica sizes surpass that of the $(t = 1, n = 5)$ configuration, which was already analyzed in Section 5.6. Our specific focus lies in exploring the potential effects of augmenting the initial replica set with supplementary spare replicas, referred to as Δ-replicas, in situations where the value of t remains fixed. The overall aim is to ascertain whether such an augmentation strategy can yield observable improvements in latency for clients dispersed across diverse geographic locations.

Experimental Setup

We use the AWS infrastructure and adhere to the same overall setup for conducting our measurements as outlined in Section 5.6. However, we expand the scope by introducing additional *regions* for replica placement, as detailed in Table 5.15a.

Here, for both $t = 2$ and $t = 3$ we begin with an *initial replica set* and then incrementally augment it by adding spare replicas which serve as Δ-replicas to improve the latency of the system. Δ-replicas are joined into the system configuration in the order in which they appear in Table 5.15a. For each system configuration that is yielded, we re-deploy all clients and replicas and subsequently start our micro-benchmark suite *after* AWARE optimized the system configuration.

Note that even for the initial replica set employs a $\Delta = 0$ configuration, and in which all replicas have the weight 1, AWARE still optimizes this configuration through the selection of an optimal leader, for the given network environment. Moreover, AWARE is configured to attempt self-optimization after every 500 consensus instances. Furthermore, we distribute 5 clients across different AWS regions, namely in *Mumbai*, *São Paulo*, *Paris*, *Ohio*, and *Sydney* to observe latencies from different system sites. All clients start sending their requests at the same time. Each system environment is measured by all clients that first send 500 requests for a warm-up phase to make sure AWARE has adapted to a fast-performing configuration. Afterward, each client measures its observed request latency as the median of a sample of 1000 requests that have been sent during the benchmark execution.

Experimental Results

We display our results in Figure 5.15b and Figure 5.15c for the $t = 2$ experiments with system sizes between 7 and 11 replicas and the $t = 3$ experiments with system sizes between 10 and 14 replicas, respectively.

In the $t = 2$ experiments, we can see that adding *Virgina* as a spare replica improves the average request latency across all clients slightly (from 553 ms to 505 ms), and subsequently joining the *Ireland* replica in the system decreases the average request latency down to 337 ms. We noticed further that by joining even more replicas (yielding $\Delta = 3$ or $\Delta = 4$ configurations) no substantial latency reductions were obtained.

In the $t = 3$ experiments, we observe that when we joined *Frankfurt*, the average request latency across all clients decreases from 619 ms to 541 ms. Yet in this ($\Delta = 1$) configuration, the latency that the Mumbai client observes slightly increases. This is because Mumbai was previously the leader in the $\Delta = 0$ configuration, but another replica was nominated leader in the $\Delta = 1$ configuration, and co-residency with the leader improves the latency of a client. As we continue to join in more replicas, the average request latency across clients further decreases: 537 ms (adding *Virginia*), 419 ms (adding *Ireland*) and 402 ms (adding *Frankfurt*).

We conclude that adaptive weighted replication can decrease the latency of geographically scalable BFT consensus, as assigning high voting weights to fast replicas supports the emergence of fast quorums in the system. Fast consensus is beneficial for clients in different sites of the systems, yet, other factors may also exist, e.g., co-locality with the leader replica.

5.8 Integration of AWARE into the HLF Blockchain Platform

In an effort to demonstrate the practical utility of AWARE, we aim to showcase its applicability as a consensus primitive within distributed ledger infrastructures. As an use case, we integrate AWARE into the Hyperledger Fabric (HLF) (Androulaki et al. 2018) blockchain framework, effectively using it as an adaptive ordering service. This service's responsibility is repeatedly achieving consensus on the subsequent block to be appended to the blockchain. AWARE may be fitting as consensus machinery within blockchain infrastructures that display the following characteristics:

- *geographic decentralization*: Typical for many blockchain systems is the dispersion of ordering nodes across diverse regions, e.g., being distributed on a planetary scale.
- *BFT*: Most blockchain systems strive to maintain their resilience even in the presence of malicious nodes.
- *adaptivity*: The protocol is adaptive and can respond to environmental changes, thus monitor, adjust and optimize its behavior to align with changing conditions.

With the integration, we want to show the practical utility of AWARE in modern (permissioned) blockchain systems.

5.8.1 AWARE as Ordering Service

Since AWARE is built on top of BFT-SMaRt/WHEAT and assumes the same interfaces, we can employ the existing integration of BFT-SMaRt/WHEAT into HLF v1.4 (Sousa, Bessani, and Vukolić 2018) which is briefly summarized in Section 2.3.4. As soon as AWARE ordering nodes are deployed, they can receive *envelopes* from clients. These envelopes contain read and write sets of transactions, (see the HLF transaction flow in Section 2.3.4). The ordering nodes run consensus instances, where in each instance they decide a batch of such envelopes, thus producing a total ordered sequence of envelopes. A *block cutter* iteratively groups a fixed number of envelopes obtained from this sequence into a *block* which is signed and broadcast to all endorsing and non-endorsing peers. As soon as a BFT proxy gathers sufficiently matching and verified messages for some block, it is committed to the ledger. In contrast to BFT-SMaRt, AWARE runs its monitoring mechanism during the ordering procedure and periodically attempts to self-optimize i.e., improve the latency of the ordering procedure by tuning its system configuration.

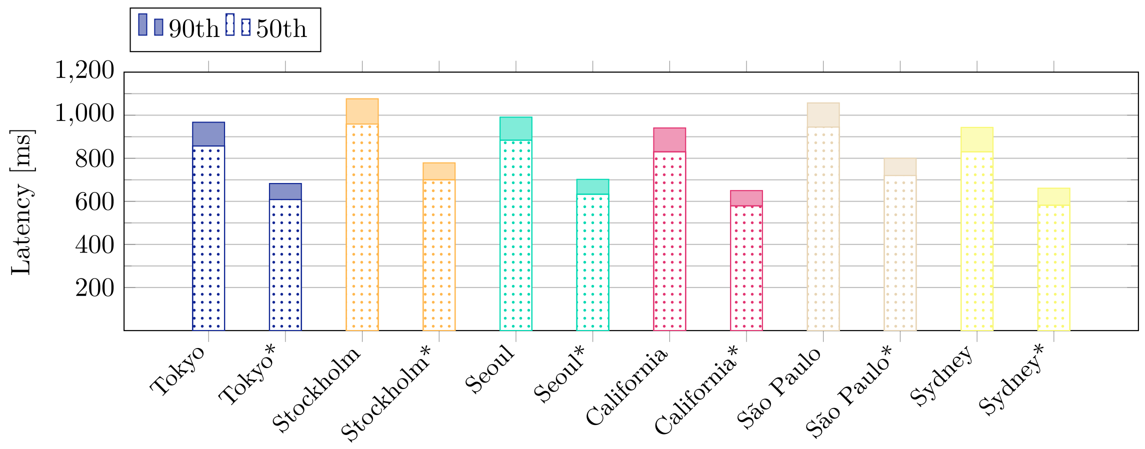
5.8.2 Evaluation

In our conducted experiments, our primary objective is to assess the performance of the AWARE ordering service within the Hyperledger Fabric framework. Specifically, we are focused on examining the time delays associated with various aspects of the ordering process, including envelope ordering, block generation, and the reception of these blocks. This analysis is conducted by monitoring the behavior of frontends situated across different AWS regions.

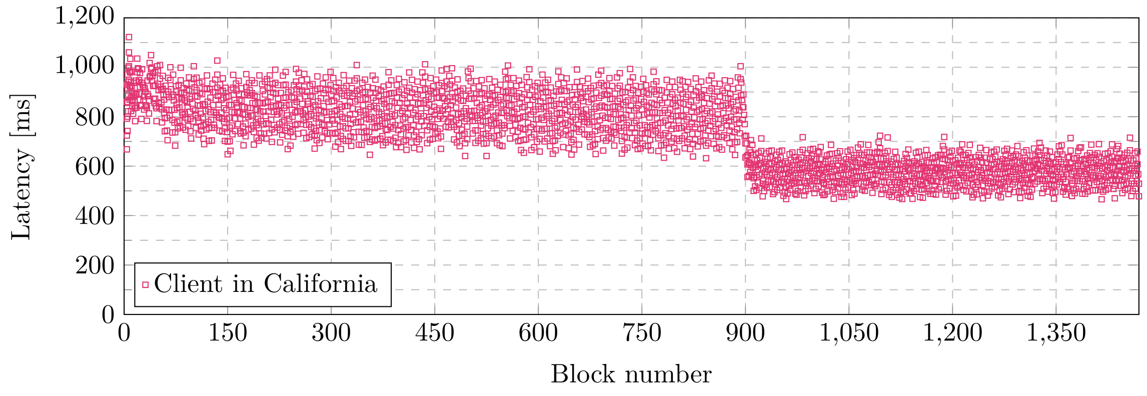
Experimental Setup. To carry out our evaluation, we adopt a configuration with parameters $t = 1$ and $\Delta = 1$. Our choice of AWS regions for deployment includes *Sydney (leader, v=2)*, *São Paulo (v=2)*, *California (v=1)*, *Tokyo (v=1)*, and *Stockholm (v=1)*, where each location is equipped with a *t2.micro* instance. These instances serve a dual purpose, operating as both ordering nodes and frontends. Additionally, an extra frontend is set up in *Seoul*. The frontends generate sufficiently many envelopes to sustain a throughput level of at least 100 envelopes per second in the system. Envelopes consist of transaction message payloads with a size of 100 bytes, as well as ECDSA endorsement signatures of 200 bytes and a 40-byte SHA-256 hash.

The ordering service is configured to create blocks containing 10 envelopes and distribute created blocks to all frontends. Note that a consensus instance may contain multiple blocks. The AWARE ordering nodes use a calculation interval of $c = 250$ consensus instances, where after every c consensus instances they attempt to optimize the system configuration.

Experimental Results. The envelope latencies, as measured by all frontends, are depicted in Figure 5.16a. Approximately around the point when block 900 is reached, the first AWARE reconfiguration takes place at consensus instance $i_{reconf} = 250$. It is important to distinguish between blocks and consensus decisions, since a single consensus can decide multiple blocks. Moreover,



(a) Latency across frontends before and after* optimization.



(b) Frontend in California.

Figure 5.16: AWARE as self-adapting ordering service for Hyperledger Fabric leads to improved latency observed by frontends.

AWARE checks for the need of a reconfiguration after every 250 consensus instances, and the reconfiguration is conducted only if the current configuration is found to be too slow.

During this phase, the AWARE system undergoes a reconfiguration, strategically relocating the maximum voting weight v_{max} from *São Paulo* to *California*. The operational behavior as perceived from *California's* standpoint is illustrated in Figure 5.16b.

This strategic optimization has a positive impact on latency across all frontends, which is evident from the improvements shown in Figure 5.16a depicting median and 90th percentile request latencies. For instance, the median latency experienced by *Sydney* (where the replica is still the leader) reduces from 830 ms to 579 ms, displaying a speedup of 1.43× as a result of this improvement.

AWARE anticipated that the reconfiguration lead to a reduction in consensus latency, projecting a decrease from 323 ms to 180 ms. Yet, it's worth noting that the improvements in latency observed by frontends exceed this reduction in consensus latency. This discrepancy arises due to the fact that an envelope submitted to the ordering service might not be promptly proposed by the leader but is delayed. The delay occurs because an ongoing consensus instance must first be completed before the envelope can be proposed in a new consensus. Consequently, there is a variable waiting period for an envelope when arriving at the leader, with the expected duration of this wait being approximately half of the consensus latency.

5.9 Additional Related Work

In this section, we present an overview of research related to AWARE. While Section 3 already summarizes most of this related work, here we aim to work out the concrete *key differences* between AWARE and related approaches.

Optimizing SMR Protocols for WANs

A variety of related research work touches on the field of optimizing SMR protocols for wide-area network deployments. An overview of these works can be found in Chapter 3.1.

EBAWA (Veronese et al. 2010) is a protocol that improves SMR latency in WANs using a hybrid fault model. It requires a trusted component on the replicas which then allows to reduce the number of replicas in the system to $2t+1$ and the communication steps needed for agreement to 2. Further, it also uses a rotating leader technique where clients send their requests to their local server. *Egalitarian Paxos* (Moraru et al. 2013) allows all replicas to propose and employs a mechanism for solving conflicts if operations interfere. Clients choose a fast-connected replica to submit their operations to it. Moreover, there is *GeoPaxos* (Coelho and Pedone 2018) which decouples order from execution and employs partial order instead of total order and exploits geographic locality to achieve fast geographic SMR.

Both Egalitarian Paxos and GeoPaxos assume systems that only tolerate crash faults. A key design difference is that AWARE assumes the Byzantine fault model, in which implementing SMR is notoriously more complex.

Mencius (Mao, F. P. Junqueira, et al. 2008) aims to optimize latency in WANs by employing a rotating leader scheme in which clients submit requests to the geographically nearest replica, using it as the leader. In contrast to AWARE, *Mencius* only supports the CFT model because of its skipping technique. The idea of a rotating leader in *Mencius* is later extended through the *RAM* protocol (Mao, F. Junqueira, et al. 2009), which adds attested append-only memory and assumes a *mutually suspicious domains* model to improve SMR latency. In this model, a client assumes the leader replica in its own geographic domain is always correct.

Thus, AWARE also makes different assumptions than Mencius (a CFT protocol) and RAM. When compared to the latter, there is no need to assume a client has a trusted (i.e., *correct*) replica in its local region that can act as a leader or that replicas are equipped with attested append-only memory.

Steward (Amir, Danilov, et al. 2010) proposes a hierarchical architecture to scale BFT replication in WANs. In every site of the system, a dedicated *group* of replicas runs BFT agreement (considerably fast since replicas are geographically close to each other). Finally, groups are connected over a crash fault-tolerant (CFT) protocol. Similar to AWARE, Steward employs additional spare replicas but with much higher replication costs since at least 4 replicas are required in every geographic system site. A more recent hierarchical approach to improve performance and scalability in geographic BFT replication is Orion (Yahyaoui et al. 2024). In Orion, local (geographically close) replication groups form *clusters*, running a BFT protocol (i.e., HotStuff (Yin et al. 2018)), where each cluster has a representative in the global group. Local pre-ordering can resolve conflicts on a cluster's local objects without global interaction (i.e., if only accessed by clients inside the same cluster). Further, a consistent broadcast (Bracha 1987; Bonomi et al. 2021) is used to disseminate blocks (containing both cross-cluster transactions and local-cluster transactions for improved durability) to all other clusters, and the global group uses the Damysus protocol (Decouchant, Kozhaya, et al. 2022) to achieve global consensus on a super-block.

In contrast, AWARE assumes a system in which all replicas are connected over a flat hierarchy instead of the two-layered hierarchy that is proposed by Steward or Orion. This may have two advantages: First, the model employed by AWARE can be easily adapted by all kinds of systems, while in Steward (and Orion) there needs to be a group of at least 4 replicas in each region, which could lead to undesired high replication costs (or impossible deployment scenarios if there are regions with too few replicas to form a cluster). Second, Steward assumes that the Byzantine replicas are uniformly distributed across all regions, but its correctness may become violated if in a single region (cluster), all replicas become Byzantine (since the geographic regions are connected over a CFT protocol in Steward). Orion makes similar assumptions and can tolerate entire cluster failing by crashing but the number of Byzantine replicas in each cluster C is limited to $t = \lfloor \frac{n_C - 1}{3} \rfloor$. In practice, it is possible that a single malicious entity gains control over all replicas deployed in the same data center (and cluster). In contrast, AWARE tolerates such failure scenarios as long as the overall number of faults is within a specified threshold, independently of which concrete replicas are affected.

WHEAT (Sousa and Bessani 2015) presents a design that employs weighted replication to lower latencies for geographic SMR (for summary see Chapter 3.4). While weighted replication was originally proposed and studied over 40 years ago (Gifford 1979; Garcia-Molina and Barbara 1985; Pâris 1986), WHEAT was the first BFT SMR protocol that assigned voting weights to improve latency in a WAN. WHEAT was crafted after empirical studies that evaluated the usefulness of different optimization techniques in practical WAN replication (i.e., tentative execution, fast execution or speculative execution).

An open challenge in WHEAT has been the question of how to select an optimal system configuration to minimize latency, and how this self-optimization procedure can be automated. This gap is addressed in this thesis by introducing AWARE, which continuously strives for latency improvements at runtime and adapts to varying network characteristics, thus optimizing its system configuration toward the environment it is being deployed.

Adaptive BFT Protocols

Moreover, adaptive BFT protocol designs have been studied by several related research works. Adaptive BFT protocols can optimize the performance of the system given varying conditions, for instance, prevent performance degradation under a faulty leader (Aublin, Mokhtar, et al. 2013), client workload (Bahsoun et al. 2015), or network bandwidth (Chiba et al. 2022) to name a few examples. A comprehensive overview of adaptive BFT approaches is given in Chapter 3.2.

In contrast to these BFT protocols which have different goals, AWARE aims to reduce latency in plantary-scale networks by adapting a WHEAT system configuration towards perceived network characteristics.

ARCHER (Eischer and Distler 2018) is a BFT protocol that uses latency-aware leader selection in geo-replicated systems. ARCHER utilizes clients' observed end-to-end response latencies to choose the leader and thus can adaptively adjust to varying workloads.

Compared to ARCHER, AWARE measures replica-to-replica latencies and uses weight tuning in addition to leader selection. AWARE follows the empirical observations of WHEAT, in which a mechanism that makes clients closer to their leaders (or using a leader in the same region) gives fewer latency improvements than just using the fastest replica as the leader (Sousa and Bessani 2015).

Adaptiveness can also be used to improve the resilience of a BFT SMR system as shown by ThreatAdaptive (D. S. Silva et al. 2021). ThreatAdaptive can adjust its fault threshold parameter t (and system size n) to shield against an perceived increase in adversarial strength (i.e., compromised replicas). When the perceived threat level is low, the ThreatAdaptive remains in or reverts to a more optimized system configuration, as a smaller system size generally enhances performance.

In contrast to ThreatAdaptive, AWARE assumes a fixed-sized replica group and can not adjust the threshold parameter t. The overall goal of AWARE's adaptation is different, i.e., tuning performance for a given, fixed-sized system in a WAN deployment by adapting voting weights and leader position. In AWARE, the system administrator must define a specific fault tolerance threshold t prior to system initialization.

RBFT (Aublin, Mokhtar, et al. 2013) is mainly concerned with preventing performance degradation by observing and redundantly running agreements using other replicas as hypothetical leaders. This way, it can be detected if the agreements under the current leader perform slower than they would under alternative replicas.

In a similar fashion, AWARE lets non-leader replicas broadcast a DUMMY-PROPOSE message which is used for measurements. Using the measurements, AWARE's underlying performance model can also detect bad leaders and switch to a more optimal leader. In contrast to RBFT, AWARE's measurement methodology is more lightweight, since not the whole agreement pattern is run redundantly. Rather than that, the prediction model is used to evaluate system configurations. Note that, the main latency benefit of AWARE in WANs comes from its use of weighted replication, which is not employed in RBFT.

BFT Consensus in Blockchains

Furthermore, many research efforts have been made to improve and tailor BFT consensus protocols towards a blockchain use case (Miller, Xia, et al. 2016; Kogias et al. 2016; Gilad et al. 2017; Kokoris-Kogias et al. 2018; P. Li, G. Wang, Chen, and Xu 2018; J. Liu et al. 2018; Yin et al. 2019; Losa et al. 2019; Bessani, Alchieri, et al. 2020; Arun and Ravindran 2020; Crain et al. 2021; Sui et al. 2022; Zhao et al. 2024).

Several BFT consensus protocols have been developed within the domain of blockchains, driven by the recognition that crafting a *"one size fits all"* solution is likely unattainable (Singh et al. 2008; Androulaki et al. 2018; Duan et al. 2018). BFT consensus protocols have been tailored to fit specific purposes and thus may differ in their assumptions and ambitions (Cachin and Vukolić 2017; Bano, Sonnino, Al-Bassam, et al. 2017; Berger and Reiser 2018a; Rüsch et al. 2019a). To illustrate this, Algorand (Gilad et al. 2017) improves the protocol scalability towards a high quantity of replicas (their experimental evaluation considers up to 500k replicas) while FastBFT (J. Liu et al. 2018) assumes and uses trusted hardware components to achieve low latency commitments. Further, the HoneyBadgerBFT (Miller, Xia, et al. 2016) can tolerate arbitrary network failures, as it adopts asynchronous atomic broadcast. Moreover, Gosig (P. Li, G. Wang, Chen, and Xu 2018) can withstand adversarial network conditions on a wide-area network.

It is noteworthy that in related work, the introduction of voting weights can also serve different purposes than optimizing performance, such as expressing *reputation*, and strengthening the robustness of a system against attacks as done by RepuCoin (Yu et al. 2019) which improves the longevity of blockchain systems.

Apart from these highly-scalable permissionless blockchain systems, research also investigates on *permissioned* blockchain systems, where often only tens of replicas are assumed to form a consortium, i.e., consisting of industrial partners and intermediaries. For this purpose, Hyperledger Fabric (Androulaki et al. 2018) has been proposed as a highly modular and industrial-grade blockchain solution. To respect the variety of the design space of possible solutions to achieving consensus, the Hyperledger Fabric blockchain platform employs modularity and encapsulates consensus as a separate building block called *ordering service*, making it exchangeable, so system developers can ideally choose the protocol that fits best for an envisioned use case.

In this thesis, we experimented with AWARE as an ordering service for Hyperledger Fabric, to showcase that it can decrease latency in a global setting and could be a potentially appealing choice to be used in a real-world permissioned blockchain use-case where Byzantine faults are assumed.

5.10 Concluding Remarks

In the field of planetary-scale Byzantine consensus systems, there is a potential advantage in employing dynamic self-optimization methods. This thesis presents a method to create an adaptive approach by using WHEAT as the weighting scheme and BFT-SMaRt as the replication protocol. The constructed protocol, AWARE, introduces continuous system measurements and the periodic calculation of an optimal configuration. Additionally, AWARE introduces a deterministic self-optimization algorithm that empowers the system to reduce its consensus latency, resulting in quicker responses to clients.

As a direct result of this thesis, a prototype implementation of AWARE has been published open-source[4]. AWARE builds upon the concept of weighted replication and adds the necessary automation to adapt to changes in environmental conditions. The proposed framework establishes robust and adaptive Byzantine consensus, automating the adjustment of voting weights and leader placement. As a result, AWARE strives for runtime improvements in latency by selecting configurations that perform well according to the WHEAT scheme, thereby solving an open deficit that existed in WHEAT as WHEAT needed to be configured manually, and could not change its configuration during the lifetime of the system.

Through comprehensive evaluations conducted across various AWS EC2 regions, AWARE demonstrated significant potential for enhancing both latency and throughput. Notably, the best configuration achieved an average speedup of approximately 38.7% in terms of observed latency across clients' sites when compared to the median configuration.

In the context of a blockchain use-case that may prioritize a high level of decentralization, the constraint of geographical scalability becomes increasingly important, especially when replicas are dispersed globally. This thesis shows that AWARE is effective even with larger system sizes, and can be also incorporated as an autonomous and adaptive consensus primitive for blockchain infrastructures such as Hyperledger Fabric.

Insights derived from AWARE have been influential for several subsequent research works: To begin with, Nischwitz et al. apply a prediction scheme that is based on AWARE towards the HotStuff protocol, thereby improving its performance (Nischwitz et al. 2022). Moreover, Köstler et al. propose a location-aware SMR protocol called Fluidity, which also uses a scheme of constant self-monitoring similar to AWARE, but optimizes *deployment decisions*, i.e., it optimizes latency during runtime by bringing replica locations geographically closer to client request origin thus adapting to changing client workloads[5] (Köstler et al. 2023). Furthermore, the next chapter of this thesis introduces FLASHCONSENSUS, which is an extension of AWARE that allows it to also adapt its resilience threshold to accelerate consensus through the tentative use of compacter consensus quorums.

[4]You can find the open-source implementation of AWARE at `https://github.com/bergerch/aware`.
[5]In Fluidity, the workload is assumed to change as client activity is affected by the change of daytime.

Chapter 6
FlashConsensus

In the previous chapter, we introduced the design of the Adaptive Wide-Area REplication (AWARE) protocol, which achieves latency improvements by tuning its weight configuration (and thus quorum system) to make the most of the heterogeneous network link latencies that it monitors. The performance benefits that can be achieved through its use of weighted replication depend on an initially chosen parameter Δ that defines the number of spare replicas in a system, which do not count for improving resilience.

In this chapter, we present insights that were partly published in a previous research article:

Christian Berger, Lívio Rodrigues, Hans P. Reiser, Vinicius Cogo, and Alysson Bessani (2024). "Chasing Lightspeed Consensus: Fast Wide-Area Byzantine Replication with Mercury". In: *The 25th ACM/IFIP International Middleware Conference*. ACM,

We introduce FlashConsensus which amplifies AWARE through the use of an adaptive resilience threshold that allows the system to autonomously switch between two discrete operational modes: (1) a *fast* mode and (2) a *conservative* mode.

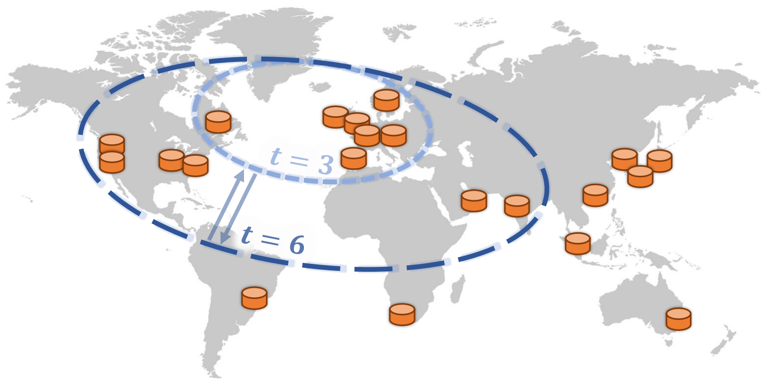
While FlashConsensus can drive consensus decisions significantly faster when running in its fast mode, the conservative mode functions as a backup that becomes effective when the system encounters less favorable conditions, in particular when the number of faulty replicas present in the system exceeds the number which the fast mode can tolerate.

Latency improvements can be achieved because FlashConsensus uses the *adaptive weighted replication* method of AWARE to autonomously assign large voting weights to a clique of fast, well-connected replicas, which in turn, form compact quorums. These compact quorums can be gathered more swiftly, and thus decrease consensus latency.

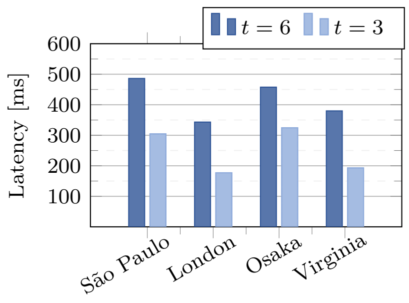
It is important to remark that even when leveraging such compact quorums, which were originally constituted for a lower resilience threshold (the one used in *fast* mode), FlashConsensus upholds the standard SMR liveness and safety guarantees (see Section 2.3.1) under optimal resilience threshold $t = \lfloor \frac{n-1}{3} \rfloor$ by virtue of its judicious use of two operational modes and integrated BFT forensics techniques. In this light, we explore new principles because FlashConsensus integrates BFT forensics (Sheng et al. 2021) as a powerful mechanism to *detect and counter attacks* such as equivocations conducted by Byzantine replicas.

How Smaller Quorums Improve Latency

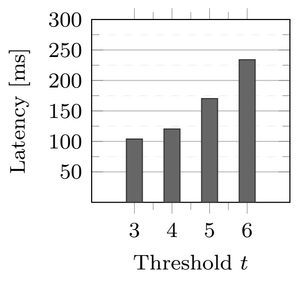
Figure 6.1 provides a visual representation of how a geographically distributed BFT SMR system can be accelerated by probing a smaller quorum of replicas. This scenario involves a weighted quorum-based system (Sousa and Bessani 2015) comprising $n = 21$ replicas positioned across 21 AWS regions (as depicted in Figure 6.1a). In a configuration tailored towards maximum resilience, the system can withstand up to $t = 6$ Byzantine replicas (the largest integer that adheres to $t < \frac{n}{3}$) and maintains $\Delta = 2$ spare replicas. Comparatively, the smallest weighted quorum, denoted as Q_v^6, encompasses 13 replicas (refer to §3.4 for detailed calculation methodology), differing by just



(a) Quorums sizes of compact quorums in WHEAT/AWARE for the resilience thresholds $t = 6$ and $t = 3$, respectively.



(b) Request latencies measured by clients in different regions.



(c) Consensus latency *vs.* resilience threshold.

Figure 6.1: Effect of varying the resilience threshold (t) towards size of compact quorums and the resulting AWARE system latency in our $n = 21$ AWS setup (as detailed in Section 6.5).

one replica from the non-weighted replication approach in which the quorum size is 14 (in general: $\lceil \frac{n+t+1}{2} \rceil$). Alternatively, if we configure AWARE to tolerate just $t = 3$ faulty replicas, then the smallest weighted quorum Q_v^3 consists of merely 7 replicas and $\Delta = 11$ spare replicas. Let us recall that AWARE can optimize this quorum to be composed of closely situated replicas, allowing for swifter vote exchanges among them and thereby expediting the consensus protocol stages (as depicted in Figure 6.1c). Ultimately, this acceleration leads to reduced end-to-end latency as observed by clients dispersed globally (as shown in Figure 6.1b).

The Need to Master the Performance-Resilience Trade-Off in AWARE

While the above-mentioned latency improvements can be achieved with AWARE if we run it with a less resilient configuration ($t = 3$ instead of $t = 6$), the configured resilience threshold remains fixed during the entire lifespan of the system and makes it more vulnerable to attacks (since fewer replicas need to fail for a safety violation than would under the optimal resilient configuration).

As a consequence, systems operators are confronted with a trade-off situation between choosing either a higher degree of resilience or having a faster system. The main two open challenges present in AWARE can be viewed as that (1) the resilience threshold and thus quorum composition rules can not be changed later on during runtime and needs to be selected manually, and (2) AWARE has no perception of a "threat level" within the system, so it can not make a sound decision on which threshold should be used. Table 6.1 summarizes the notations that we use in this chapter.

Table 6.1: List of notations.

Notation	Definition
n	number of replicas in the system
f	number of Byzantine replicas ($f \leq t$)
t	resilience threshold for the BFT protocol ($t = \lfloor \frac{n-1}{3} \rfloor$)
t_{fast}	resilience threshold used in the fast mode ($t_{fast} = \lceil \frac{t}{2} \rceil$)
$Q^t_{[v]}$	a ([v]ote-based weighted) quorum for threshold t

The Essence of Threshold-Adaptive BFT SMR

The tentative use of a lower resilience threshold t_{fast} seems to challenge the fundamental safety guarantee of BFT SMR at first glance. This is because any quorum system designed to withstand such a lower resilience threshold $t_{fast} < t$ could be potentially exploited by a Byzantine adversary during the system's lifespan as soon as the number of actual faulty replicas f exceeds this threshold, such as when $t_{fast} < f \leq t$. In this case, the assumption of all quorums intersecting pairwise in at least some correct replica no longer holds, and conflicting decisions appearing in two correct replicas' decision logs becomes a possibility.

A decisive concept in our research is the innovative utilization of *BFT protocol forensics or accountability* (Shamis et al. 2022; Sheng et al. 2021). Instead of solely employing it as a post-incident investigative tool, we repurpose it as a preemptive defense mechanism against Byzantine attackers. Traditional usage of BFT protocol forensics assumes that discrepancies between values are detected by the client through analysis of logged messages from replicas, identifying replicas involved in conflicting decisions. In FlashConsensus, the responsibility of identifying faulty replicas shifts to all correct replicas, enabling the system to independently identify and expel equivocating entities. Within a system designed to tolerate up to t_{fast} faulty replicas, auditing procedures can uncover instances of agreement breaches and identify $t_{fast} + 1$ faulty replicas, assuming there are no more than $2t_{fast}$ Byzantine replicas present (a result of (Sheng et al. 2021)). By setting $t_{fast} = \lceil \frac{t}{2} \rceil$, we ensure that audits are consistently effective under the optimal resilience ($f \leq t = \lfloor (n-1)/3 \rfloor$).

The periodic auditing of replicas can be used to construct a recovery mechanism: When inconsistent decisions (called *equivocations*) in the decision logs of correct replicas are recognized, at least $t_{fast}+1$ Byzantine replicas that participated in two quorums causing the equivocation can be identified and removed from the system. Subsequently, all replicas roll back to a last stable checkpoint (which certifies a consistent decision log up to some index) and then re-apply a consistent sequence of decisions as defined in the regency of a new leader within the conservative mode. Interestingly, the auditing mechanism not only serves the purpose of threat-level awareness (facilitating the recovery procedure) but also acts as a *determent to Byzantine adversaries* because a Byzantine adversary loses a significant portion of its controlled replicas ($t_{fast} + 1$) when attempting an equivocation.

The previously explained rollback mechanism must be used with caution, otherwise, SMR safety is endangered. In particular, we must ensure that no side-effect of a rollback is ever observed by clients, e.g., a client operation must only *complete*, if the operation never becomes undone (i.e., due to a rollback of decisions), and the finality of operations holds forever (an operation is final if it is unchangeably assigned into a specific slot i in the decision log with all slots $j < i$ having operations unchangeably assigned as well). As a result, it becomes necessary to revisit and adjust the criteria for matching replies expected from clients. This ensures that clients can ascertain if an operation is *finalized* by the system, thereby upholding consistency (i.e., linearizability (Herlihy and Wing 1990)) and liveness as in standard SMR (Castro and Liskov 1999).

Striving to minimize consensus latency and subsequently making clients await a *higher-than-usual* number of responses from various globally dispersed locations — all in the interest of upholding linearizability — contradicts the primary objective of reducing end-to-end request latency. Consequently, FlashConsensus extends the programming model of standard BFT SMR by integrating the concept of *correctables*. These enable client applications to employ incremental consistency guarantees (Guerraoui, Pavlovic, et al. 2016), thus simplifying and encapsulating the speculation on the client side. Our experiments demonstrated that by adopting this abstraction, clients can achieve an even stronger reduction in their observed request latency.

Finally, we must pay attention to the aspect of *liveness*. Ensuring the liveness of SMR implies that requests initiated by correct clients must eventually complete successfully (see Section 2.3.1). In optimistic executions in which quorums are collected using the threshold t_{fast}, liveness may become violated when a Byzantine adversary controls $f > t_{fast}$ replicas. The group of Byzantine replicas may try to block the system by omitting to respond to clients or by not participating in the agreement pattern, thus preventing quorums from being collected. To resolve liveness issues, FlashConsensus relies on its second mode of operation, the *conservative* mode, which succeeds the protocol execution using a quorum system with optimal resilience, withstanding t faulty replicas.

6.1 System Model and Preliminaries

In this section, we briefly explain a few preliminaries, in particular by explaining our use of the BFT forensic techniques (Sheng et al. 2021).

System Model. We assume the same system model like in AWARE as explained in Chapter 5.1, assuming a partially synchronous system, a number of Byzantine replicas that is always less or equal to $t = \lfloor (n-1)/3 \rfloor$, and an arbitrary number of Byzantine clients. All Byzantine entities may display arbitrary behavior and may even collude. We assume a threshold adversary can control and orchestrate the group of Byzantine replicas. Nevertheless, no Byzantine entity can compromise cryptographic primitives. Before the system starts, every replica and client possesses a key pair consisting of a private and public key, and all public keys are well-known to everyone universally. All entities are assumed to use secure primitives to create and verify message signatures, and a secure hash function.

BFT Protocol Forensics. BFT protocol forensics (Sheng et al. 2021) shifts the focus of BFT research from tolerating faults to the aftermath, i.e., on events after erroneous behavior happened in a system. Specifically, the objective of BFT protocol forensics is to detect malicious behavior and determine which replicas deviated from the described protocol (they are the "culprits"). Successful implementation of BFT protocol forensics aims to accurately pinpoint as many malevolent replicas as possible, substantiated by indisputable cryptographic evidence termed as "proof of culpability" (PoC).

Our particular focus lies in the forensic support of PBFT and we consider the variant of PBFT when implemented with signed messages, originally named "BFT-PK" by Castro (Castro 2000), and denoted as "PBFT-PK" in the work by (Sheng et al. 2021). We stick to the name "PBFT-PK". In the concluding stages of the Prepare and Commit phases, replicas collect *signed messages* to form the *quorum certificates* (QC) for each phase. To optimize performance (i.e., bandwidth usage or required storage), it is possible to replace a set of signed messages by a single aggregate-signature as done in (Sheng et al. 2021). In our implementation, a QC remains a set of signed messages just as originally described by Castro et al. (Castro and Liskov 1999).

The purpose of using signatures on messages instead of message authentication codes[1] lies in the *non-repudiation* property of signatures. Replicas can be held accountable if they engaged in the preparation or commitment of a value during a distinct view. A QC proves the "voting behavior" of a particular subset of replicas within the agreement pattern, i.e., a replica may cast its vote for a specific value inside a specific context (defined by the sequence number and view).

[1]Message authentication codes have been intended to improve performance of PBFT (Castro and Liskov 1999).

Upon the completion of the agreement, each replica transmits a response to the client. In cases where the client discovers two conflicting REPLY messages $\langle \text{REPLY}, e, v, \sigma\rangle$ and $\langle \text{REPLY}, e', v', \sigma'\rangle$ where $v \neq v'$ denote the conflicting values, e and e' are the corresponding view numbers and σ and σ' the corresponding COMMIT quorum certificates, it can then start the following forensic procedure as originally described in (Sheng et al. 2021): First, if both messages are from the same view ($e = e'$), it is sufficient to check the *commitQCs* which intersection yields the at least $t + 1$ culpable Byzantine replicas, i.e., the forensic procedure returns $\sigma \cap \sigma'$. Otherwise, the client needs to request a proof between the two views e and e' from the replicas and each replica checks its set for the NEWVIEW messages to find the smallest view $e^* > e$ such that the status certificate M includes the highest lock (e'', v'', σ_i) where $v'' \neq v$ and $e'' \leq e$ and sends it back to the client. In the status certificate M there can be either a conflicting lock in the same view (then the forensic procedure returns the intersection of two *prepareQCs* which indicate the at least $t + 1$ culprits), or else the intersection of the senders of M with the *commitQC* (signers of σ) proves the culprits.

An important result of (Sheng et al. 2021) is that PBFT forensic support can detect and identify at least $t + 1$ Byzantine replicas, as long as the actual number of faults f in the system is not greater than $2t$ and in case there are more than $2t$ Byzantine replicas, PBFT forensic support is impossible. FlashConsensus considers this limitation by choosing $t_{fast} = \lceil \frac{t}{2} \rceil$. Further, as we will see later, FlashConsensus only needs a single transcript from one correct replica to construct the *proof of culpability*. A transcript consists of a set of aggregate signatures collected over several agreements, stored and maintained alongside the decision log.

6.2 The Design of Threshold-Adaptive BFT: FlashConsensus

In this section, we introduce the blueprint of FlashConsensus, a transformation for quorum-based BFT protocols that strives for ongoing self-improvement during runtime. It accomplishes this by flexibly adjusting the resilience threshold and fine-tuning the weights assigned to replicas. This adaptation facilitates the emergence of smaller and faster consensus quorums, leading to the ordering of operations with minimal latency. The idea of FlashConsensus strives to harmonize two crucial objectives. First, it proposes to deploy and run the BFT SMR system under optimal resilience with $t < \lfloor \frac{n-1}{3} \rfloor$, and second, it strives to accelerate consensus decisions in common scenarios when there are only a few failures (i.e., less than $\lceil \frac{t}{2} \rceil$). To make this optimization work, FlashConsensus needs to achieve threat-level awareness.

6.2.1 The Goal of FlashConsensus

The ambition of FlashConsensus is to facilitate the construction of fast BFT SMR variants for planetary-scale deployments, which fulfill both SMR safety and liveness for the optimal, $t = \lfloor \frac{n-1}{3} \rfloor$ resilience bound and and continuously optimizes itself to obtain fast commit latency in the expected common case when there is a stable, correct leader and no more than $t_{fast} = \lceil \frac{t}{2} \rceil$ faulty replicas.

6.2.2 The Two Modes of Operation

This twofold strategy is realized through the utilization of two operational modes, as done in approaches that aim for enhancing performance (Aublin, Guerraoui, et al. 2015) and optimizing resource efficiency (Distler et al. 2015). The system initiates its operations in the *conservative* mode wherein it engages in running consensus protocol instances using quorums that are capable of withstanding up to t failures.

At regular intervals, specifically following a predefined number of consensus instances, the system attempts to switch to the *fast* mode, where the resilience threshold is constrained to t_{fast} failures. As long as the leader remains correct and the actual failure count f does not exceed t_{fast}, FlashConsensus maintains this configuration, using smaller quorums to expedite the consensus

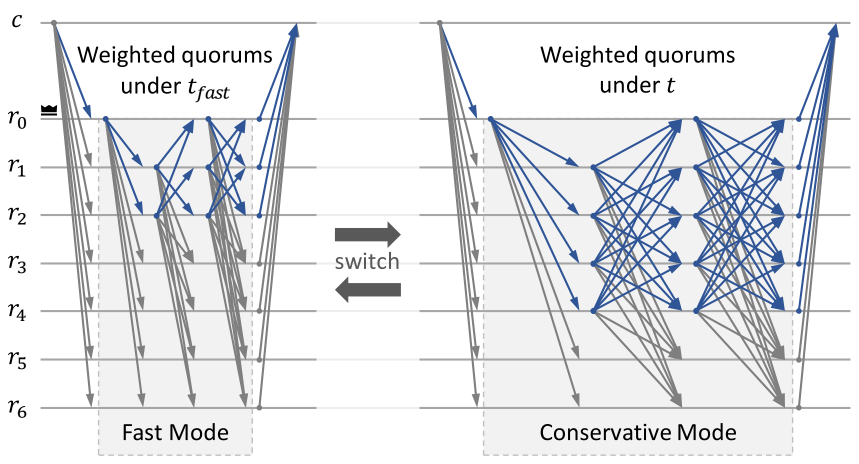


Figure 6.2: FlashConsensus two modes of operation for $t = 2$ and $t_{fast} = 1$.

instances in the expected common case. Only if the threat-level awareness of the system finds that the fast mode can not guarantee correct executions, FlashConsensus promptly reverts to the conservative mode. Reasons for this reversion include (1) the replica audits discover equivocations, (2) realized latency enhancements fall short of expectations, or (3) no progress is made. The dynamics of these two operational modes are visually represented in Figure 6.2.

6.2.3 A Note on Obtaining Smaller Quorums

As previously demonstrated, the reduction of t alone yields only moderate advantages in quorum size when using egalitarian quorum systems. For instance, as portrayed in Figure 6.3, diverse scenarios for a system comprising 21 replicas are depicted. Through the adoption of weighted replication with $t_{fast} = 3$, it is feasible to establish quorums as small as 7 replicas.

Building Block 1: Weighted Replication

Weighted replication distributes weights to replicas and enables proportionally smaller quorums through an intersection in replicas with high weight. Thus, weighted replication preserves the usual guarantees for t-dissemination quorum systems, i.e., all quorums must overlap in at least one correct server (consistency), and at least one quorum remains always accessible (availability).

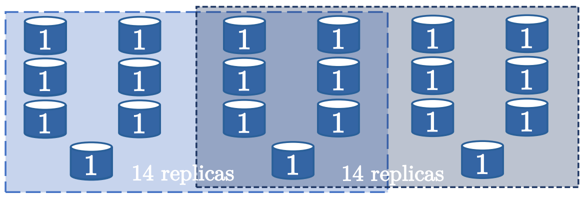
The weight distribution scheme introduced by WHEAT (Sousa and Bessani 2015) successfully meets these prerequisites, assuring the existence of a minimal quorum composed of just $2t_{fast} + 1$ replicas, irrespective of the total count n.

To comprehend the minimality of this quorum, it is necessary to recognize that the availability criterion mandates that the largest quorum within the system accommodates a maximum of $n - t_{fast}$ replicas. To fulfill the consistency requirement, this particular quorum must intersect with all other quorums by a minimum of $t_{fast} + 1$ replicas. Consequently, the most compact quorum feasible must comprise at least $2t_{fast} + 1$ replicas.

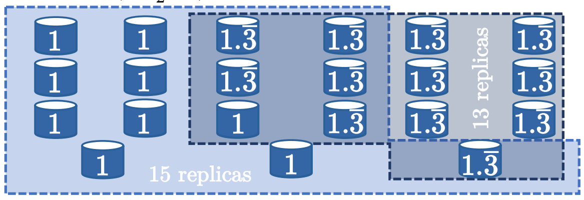
The choice of weights for the replicas in WHEAT follows a methodical approach. Specifically, $2t_{fast}$ replicas are allocated a voting power of v_{max}, exceeding 1, while the remaining replicas retain a voting weight of 1. We explain more details on WHEAT's weighting scheme in Section 3.4.

Building Block 2: Automatic Weight Re-assignments

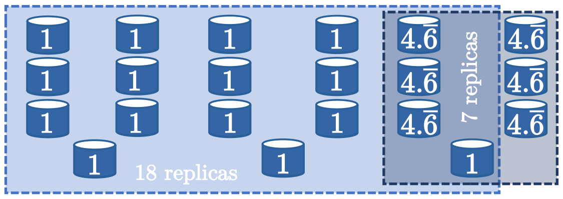
The next challenge in employing weighted replication is the automatic re-distribution of weights to optimize for system conditions at runtime. In our particular scenario, we need to choose $2t_{fast}$



(a) *Egalitarian ($t = 6$)*: All quorums have the same size of $\lceil \frac{n+t+1}{2} \rceil$ replicas.



(b) *Weighted ($t = 6$)*: A quorum contains at most $n - t$ and at least $2t + 1$ replicas.



(c) *Weighted ($t = 3$)*: Quorums can become *small* by intersecting in *fast* replicas.

Figure 6.3: Overview over BFT quorum systems for $n = 21$ replicas.

well-connected replicas to hold v_{max} voting weight. For this purpose, we can leverage the latency monitoring, prediction model, and weight reassignment procedure of AWARE (see Chapter 5) and shift the focus to the challenge of running the overall protocol optimistically, i.e., tolerating a few failures. We will accomplish this by starting an instance of AWARE that is configured with a resilience threshold of t_{fast}, which will automatically lead to our desired quorum system that enables small quorums and the selection of suitable (i.e., fast) replicas to be the ones compromising this quorum by obtaining high voting weights through AWARE's re-distribution procedure.

Recall that the influence of malicious replicas on these measurements is bounded by AWARE's sanitization strategy (see Section 5.2.2) and the verification of a replica reported values using its geo-location, as done in secure location services (Kohls and Diaz 2022).

6.2.4 Open Design Challenges

We build our design upon the capabilities of AWARE, which grants us the features to employ small quorums and self-optimizing weights based on the measured replica-replica latencies. Yet, the envisioned design of FlashConsensus still entails multiple open challenges:

- First, we need to integrate mechanisms dedicated to identifying and diagnosing system anomalies when the number of failures exceeds the threshold in the fast mode, i.e., when $f > t_{fast}$.
- Second, a resilient reconfiguration mechanism is necessary to safely and effectively transition from the fast mode back to the conservative mode under such circumstances.
- Third, the client-replica contract must guarantee linearizability within the runtime of our adaptive dual fault-threshold strategy.

Replica

F1 **Find evidence:** Let S and S' be the two sets of replicas with diverging checkpoint digests. The auditor tries to collect signed lists of decision proofs from consensus instances $i-k+1$ to i from at least one of the replicas of each of these two sets.

F2 **Produce PoC:** When such logs are obtained, the auditor checks the logs to find the first consensus instance with diverging decisions. Once such an instance is found, the auditor checks the proofs of decisions.

1. If any of the proofs of decision is invalid, the log signed by the replica that provided it is a PoC for the replica.
2. If both proofs are valid, the auditor finds at least $t_{fast}+1$ malicious replicas that provided signed Accept messages for both decisions. These two conflicting proofs are the PoC (proof-of-culpability) for the malicious replicas.

F3 **Blame culprits:** If $poc \neq \emptyset$: Broadast a $\langle \text{POC}, poc \rangle$ message.

Figure 6.4: Lightweight forensics procedure (Berger, Rodrigues, et al. 2024).

6.3 The Algorithm

The consolidated algorithm of FlashConsensus is presented in Figure 6.5. The algorithm starts with a client submitting a request o to all replicas (C1). By **Rule** (S1), replicas start a *timer* for each request, and the leader creates a batch of unordered requests and proposes it through the normal case operation of the underlying base protocol Aware, or Aware$\star$ if the system runs in *fast* mode. The rest of the algorithm summarizes all the required extensions on AWARE to integrate the novel mechanisms of FlashConsensus. In the next subsections, we describe these mechanisms in the context of the challenge they address in our protocol design.

6.3.1 Challenge #1: Dealing with $f > t_{fast}$ Failures

We first explain how to make FlashConsensus ensure standard SMR safety and liveness guarantees for the optimal resilience threshold t.

Safety

When the system enters the fast mode, we must consider the possibility that the Byzantine adversary manipulates more than t_{fast} replicas. The group of Byzantine replicas can cause an equivocation, where correct replicas may commit different decisions in the same consensus (leading to an inconsistency in their decision log). This is because the smaller quorums of the fast mode do not ensure intersection in at least one correct replica.

Periodic Checkpoint To handle this scenario, the system must detect if the actual number of faults (f) surpasses the resilience threshold of the fast mode (t_{fast}) and, if necessary, revert the system to the conservative mode. We need to check the state of the replicas through periodic *checkpoint messages* (as in PBFT (Castro and Liskov 1999)) to ensure they are consistent and detect faults. By **Rule** S3, on every k completed consensus instance, each replica takes a snapshot of the service state and broadcasts to all replicas a signed message with the digest of this snapshot h and the highest consensus instance i that affected this checkpoint. Each replica waits for $n-t$ matching checkpoint hashes for the same consensus instance to define the checkpoint as *stable*. Should any discrepancies arise during this procedure, such as when non-matching checkpoints are observed by a correct replica (the *auditor*), the lightweight forensic procedure illustrated in Figure 6.4 is started. The purpose of this procedure is to identify possible culprits and acquire an indisputable *Proof-of-Culpability* (PoC) for the replicas that violated the protocol.

[1] In case of an equivocation, two correct replicas decided different values. This means the auditor will eventually receive logs for both values from at least one correct replica that committed each value. Replicas do not garbage collect such logs prior to achieving a stable checkpoint which requires confirmation from $n-t$ replicas.

Client

C1 **Invocation:** Send $\langle \text{REQUEST}, o \rangle$ to all replicas.

C2 **Finalization:** Accept a result *res* for o if received a set of matching replies $rep = \{\langle \text{REPLY}, h(o), \mathit{fast}, \mathit{res} \rangle\}$ such that either:

1. $\mathit{fast} \land |\mathit{rep}| \geq n - t_{\mathit{fast}} - 1$ OR
2. $\neg \mathit{fast} \land \sum_{r \in \mathit{rep}} \mathit{weight}(r) \geq 2t \cdot V_{max} + 1$.

In case of a timeout, keep re-sending the request and inspecting the decision log of replicas until one of the conditions above is satisfied.

C3 **Panic:** Broadcast $\langle \text{PANIC}, o, \mathit{rep} \rangle$ to the replicas if replies $\mathit{rep} = \{\langle \text{REPLY}, h(o), \texttt{true}, * \rangle\}$ contains diverging results for operation o.

Replica

State

fast	mode of operation	boolean	`false`
chkp	last stable checkpoint	bytes	`null`

Building Blocks

AWARE	AWARE SMR protocol (conservative)
AWARE⋆	normal case operation of AWARE in fast mode
AUDIT	lightweight forensics procedure of Figure 6.4

S1 **Request Processing:** Start a *timer* for each received client request. If leader, create a batch of requests and propose it using AWARE⋆, if *fast*, or AWARE, otherwise.

S2 **Client Panic:** If *fast* and received a message $\langle \text{PANIC}, o, \mathit{rep} \rangle$ with diverging signed replies for o in *rep* from a client, run AUDIT.

S3 **Periodic Checkpoint:** When a snapshot of the service state chkp' is created after processing consensus j, broadcast signed message $\langle \text{CHECKPOINT}, h(\mathit{chkp}'), j \rangle$. If $n - t$ matching checkpoint hashes $h(\mathit{chkp}')$ for j are received, update the last stable checkpoint *chkp* to chkp'. If there are no $n - t$ matching checkpoints, run AUDIT.

S4 **Switch:** After deciding θ consensus instances in a row using AWARE, set *fast* to `true` (the optimization interval θ is inherited from AWARE).

S5 **Abort:** If *fast*: Broadcast a VIEW-CHANGE message if one condition applies: (1) a request *timer* expires, (2) a message $\langle \text{POC}, \mathit{poc} \rangle$ with a valid PoC is received from some replica, or (3) upon *consensus latency disappointment*—see §6.3.1.

S6 **Synch. Phase:** Upon receiving $t + 1$ matching VIEW-CHANGE messages, use AWARE' synch. phase to replace the current leader and synchronize the decision log. If *fast*, set *fast* to `false` and run AUDIT if no $\langle \text{POC}, \mathit{poc} \rangle$ message has been received so far.

1. When waiting for NEW-VIEW messages, the next leader checks for PoCs and ignores messages from equivocating replicas. When consolidating operations for each position of the decision log, the new leader picks the most commonly reported prepared value.
2. Upon a PoC is produced or received during the synch. phase, all replicas roll back to *chkp* and use the decisions (with proofs) obtained from the new leader to re-execute decided operations. The new leader proposes $\langle \text{RECONFIGURE}, \mathit{culprits}, \mathit{poc} \rangle$ using AWARE.
3. Upon deciding $\langle \text{RECONFIGURE}, \mathit{culprits}, \mathit{poc} \rangle$, a replica verifies the *poc* using AUDIT, and removes the *culprits* from the system.

Figure 6.5: A summary of FlashConsensus.

This protocol can identify equivocating replicas in the system if there are no more than $2t_{\mathit{fast}}$ faulty replicas. In fact, Sheng et al. show that it is impossible to identify misbehaving replicas if there are more than $2t$ Byzantine replicas in a system tolerating t failures (Sheng et al. 2021). This limitation lead FlashConsensus use $t_{\mathit{fast}} = \lceil \frac{t}{2} \rceil$.

In case the auditor encounters $n - t$ checkpoint hashes that match during the procedure's execution, the auditor prematurely halts the lightweight forensics procedure. This action prevents Byzantine replicas from blocking correct replicas in forensics procedures since a Byzantine replica can send a non-matching checkpoint but never send its corresponding log. After completing the lightweight forensics procedure, and if a PoC is produced, the auditor broadcasts this PoC to all replicas. In this case, it means that an inconsistency has been discovered in the system and that the system needs to roll back to a consistent state. As we will see in the next Section 6.3.2, this makes the system abort to the conservative mode and conduct a reconfiguration by which the blamed replicas are expelled from the replica set.

Client Panic The lightweight forensics protocol also triggers if a client detects non-matching signed replies for a request (**Rule** C3). When this happens, the client sends a Panic message with conflicting replies to the replicas. By **Rule** (S2), a replica that receives a Panic message with correctly signed conflicting replies starts the lightweight forensics protocol, but fetches logs from the last checkpoint until the consensus instance that decided the problematic request.

Liveness

In addition to equivocations, if the adversary controls more than t_{fast} replicas ($f > t_{fast}$), these replicas can adopt a passive strategy, impairing the system's liveliness by staying *silent*. If the number of correct replicas in the system falls below $n - t_{fast}$, a crucial liveliness assumption of the consensus protocol is violated since it assumes no more than t_{fast} failures. This may result in two unfavorable situations:

1. **Silent replicas might not participate in the consensus.** Firstly, the ordering of client requests might be disrupted. This situation is easily spotted because request timers are triggered thus initiating a leader change within the system (akin to the synchronization phase in BFT-SMaRt). We will explain in the next section, that this sub-protocol reverts the system to a conservative configuration, in which up to t failures are tolerated.

2. **Silent replicas might not reply to clients.** The second situation is more intricate, as the request could be ordered by the replicas, but Byzantine replicas might only abstain from transmitting responses to the client. Subsequently, the client fails to consolidate enough responses to a request needed for finalization. A possible solution to this problem is allowing a client to indicate this situation by broadcasting a panic message to the replicas. This message could serve as a trigger for the synchronization phase and switch the system back to its conservative mode. However, this panic routine must be implemented carefully because a Byzantine client could abuse it to repeatedly force the system to always operate in conservative mode. This type of weakness is inherent to many optimistic protocols (Aublin, Guerraoui, et al. 2015).

This mechanism could make FlashConsensus optimization fragile, as a single malicious client can undermine latency improvements. Thus, we propose an alternative approach for dealing with this situation (as defined next by the *Finalization* Rule).

Finalization By **Rule** C2, if the client does not receive the required number of replies, it periodically checks the decision log until the next checkpoint to see where its operation appears in the finalized decision log. This behavior is similar to that of blockchain clients, which inspect the blockchain until their requests are contained in a block at a certain position before the blockchain head. This approach ensures that clients can benefit from FlashConsensus's low latency as long as there are no more than t_{fast} faulty replicas in the system.

Preventing Performance Degradation

The design goal of FlashConsensus is to solely *improve* the latency of the system through the tentative use of the fast mode of operation. Yet, to guarantee that FlashConsensus never leads to performance degradation when compared to the performance of its conservative mode, all replicas must periodically monitor the performance of the consensus pattern and then compare it with the expectations they have for the conservative mode.

Consensus latency expectations. Replicas derive their consensus latency expectations from the latency prediction model at the core of AWARE's architecture (see Section 5.3.2). By using this prediction model, replicas can anticipate their consensus latency in the protocol's conservative mode by employing the network latency matrix. This allows them to set a *consensus latency expectation threshold*. Should a replica determine that the currently observed consensus latency in the fast mode surpasses this threshold, they halt the ongoing consensus and request a synchronization phase by broadcasting STOP messages. When $t+1$ replicas request such synchronization phases, the entire system transitions into the conservative mode (as elaborated in Section 6.3.2). In the end, FlashConsensus tentatively employs the fast mode only if replicas observe that the consensus latency is less than the expected consensus latency in the conservative mode.

6.3.2 Challenge #2: Reconfiguration of the System

FlashConsensus needs to reconfigure between two modes of operation: *fast*, in which compact quorums can be used and t_{fast} failures are tolerated, and *conservative*, which tolerates t failures and eymploys standard-size quorums.

Switch

Initially, the system starts in the conservative mode, and by **Rule** S4 after finishing a predefined number of θ consecutive consensus instances, it switches to the fast mode. This reconfiguration is executed deterministically at a certain point in the execution, i.e., after a certain consensus instance is decided. At this point, the switch requires each replica to locally change the fault threshold to t_{fast} and re-calculate the quorum system before starting the next consensus instance.

Abort

A replica remains in the fast configuration until the underlying algorithm synchronization phase is initiated by **Rule** S5. This approach can be done deliberately due to either safety or liveness issues, as discussed in previous subsections. In both cases, we require the participation of $t+1$ replicas to start the view change, which always runs considering threshold t, not t_{fast} (which might already be violated).

Synchronization Phase

During the synchronization phase (*view change* in PBFT parlance), the newly elected leader receives from $n-t$ replicas the log of the decided instances since the last checkpoint and verifies it for diverging decisions using the lightweight forensics procedure (Figure 6.4). **Rule** S6 is applied in subsequent steps:

1. If multiple decisions for the same slot exist, the new leader selects the most commonly reported one to consolidate the decision log.
2. The first transaction of the newly elected leader's regency, after repairing the system to a single transaction history, is a reconfiguration request (Bessani, Sousa, et al. 2014). This request aims to remove the Byzantine replicas involved in the equivocation and contains the PoC generated during the forensics procedure. Correct replicas remove these compromised replicas from the system after processing reconfiguration requests with valid PoCs.
3. Further, if the replica was subject to equivocation and the new leader decided differently from it, then it might need to roll back its states to a previous *stable checkpoint* (see §6.3.1), reapplying the correct transaction history as defined by the new leader on this state.

6.3.3 Challenge #3: Ensuring Linearizability

BFT SMR frameworks maintain the *linearizability* (Herlihy and Wing 1990) of a lineaizable service implementation, which is a correctness criterion of the system (i.e. a safety property) and the

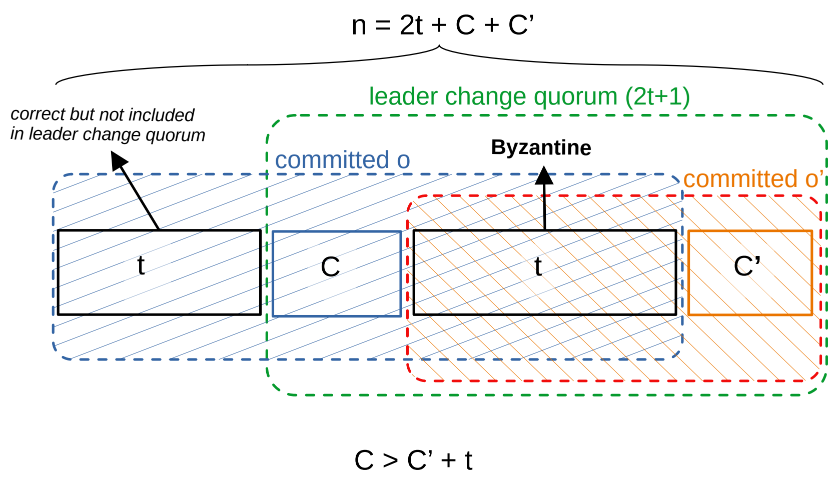

n = 2t + C + C'
leader change quorum (2t+1)
correct but not included in leader change quorum
committed o
Byzantine
committed o'
t
C
t
C'
C > C' + t

Figure 6.6: Using status information from replicas only to determine the decision log.

consistency guarantee that the clients expect. Roughly speaking, linearizability guarantees that the client does not observe system behavior other than if it would have issued its operation towards a centralized service implementation that executes all operations atomically, one after another.

In the BFT model, a client consolidates the correctness of a result by waiting for at least $t+1$ matching replies from the replicas. This holds true as long as all operations are ordered by the system. When a BFT SMR system integrates the read-only optimization this response quorum size becomes $\lceil \frac{n+t+1}{2} \rceil$ (a proof for this claim this can be found in the PhD thesis of Castro (Castro 2000)). These concrete sizes for response quorums resemble the ones which WHEAT (Sousa and Bessani 2015) uses, which can be also found in FlashConsensus *when its operates in its conservative mode*:

- $t+1$ matching *responses* consolidate a correct result if no read-only optimization is used.
- $2t \cdot v_{max} + 1$ collected *weights* of responses consolidate a correct result if a read-only optimization is used (here: a weighted *quorum* ensures intersection between any read and write).

Up to here, there is no difference to WHEAT. But if the FlashConsensus system operates in its tentative *fast mode*, then we need to rework the number of matching responses that a client requires for its consolidation as employed by **Rule** C2. This is because of equivocations that might lead to possible, divergent decisions which affects the durability of operations that have been executed after the last stable checkpoint. While such divergent decisions can be detected and resolved by FlashConsensus, it is still necessary to ensure no client would ever accept a result originating from such a divergent decision that is rolled back later.

As a small experiment of thought, suppose a client would wait for matching responses from *all of the n* replicas (ignore the liveness aspect of this modification for now). In this case, the client would be able to safely accept the operation result, because it knows that the operation was committed by sufficiently many correct replicas (here: *all*) to become durable (which would be trivial to show, since no divergent decisions could have happened). What needs to be shown is that an operation result that is accepted by any client was committed in a decision that remains durable, i.e., such a decision can always be defined as the correct decision to be permanently committed by the system.

Necessary Response Quorum when Using only Status Information from Replicas.

Figure 6.6 illustrates the scenario where two clients received replies for conflicting operations o and o' with $o \neq o'$ from two different quorums Q and Q', where both operations have been decided to the same slot i (we disregard the leader change quorum for now). We must further assume that

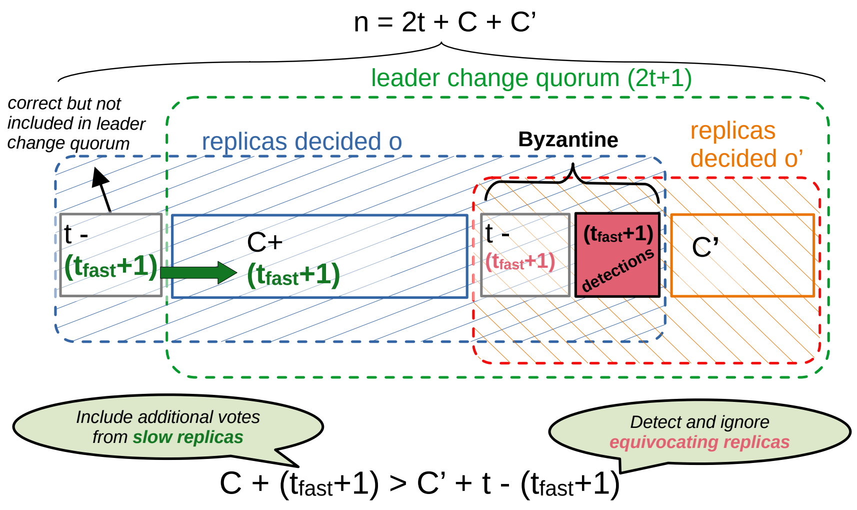

n = 2t + C + C'
leader change quorum (2t+1)
correct but not included in leader change quorum
replicas decided o
Byzantine
replicas decided o'
t - (tfast+1)
C+ (tfast+1)
t - (tfast+1)
(tfast+1) detections
C'
Include additional votes from slow replicas
Detect and ignore equivocating replicas
C + (tfast+1) > C' + t - (tfast+1)

Figure 6.7: Using status information from replicas, and the PoC from the audit to determine the decision log.

the intersection $Q \cap Q'$ contains up to t Byzantine replicas which may have responded to a client with the result of the execution of either o or o' in slot i.

Since a divergent decision always causes a leader change, we also need to ensure that a client only accepts the result of operation o if o permanently stays at position i. This means a new leader must be able to collect sufficient status information to irrefutably define o as the convergent decision for slot i, *even if some correct replicas committed* o' due to the equivocation. Note that the leader change quorum is only $2t + 1$, i.e., the leader may not collect information from the t slowest, but correct replicas (as such they may not be able to contribute their status information as shown in Figure 6.6. Further, in the set of $2t + 1$ replicas that provide status information, there might be still t Byzantine replicas that can switch sides, i.e., they may vote for o' in the leader change protocol. For the purpose of better exposition, in Figure 6.6 we assume that $n = 3t+1$ and we can write this as $n = 2t + C + C'$ (directly from the figure), where C are correct replicas, contributing to the collected status information that o is indexed at position i, and C' is respectively for o'.

In this scenario, the leader change quorum may only collect the status from $C + t + C'$ replicas, and since we know that this number equals $2t + 1$, it directly follows that $C + C' = t + 1$.

If a client wants the guarantee that operation o stays in slot i even after a potential rollback, then o must be the committed majority value (comitted by *correct* replicas) within the leader change quorum so we need to subtract t Byzantine replicas which may have replied with o but decided o' instead (and will contribute o' to the status collection during leader change). Summarizing, a client must know that $C > C' + t$ (see Figure 6.6) and since we know $C + C' = t + 1$ only the condition of $C' = 0$ satisfies the inequation. That means a client must collect $|Q| = n$ matching responses to ensure linearizability. As we see, when implementing the mechanism for picking consistent decisions after the rollback by relying on replica status information alone, *waiting for* $n - t$ *matching replies is insufficient to ensure a value is not erased during a leader change.* If we rely only on status information from replicas, the necessary response quorum that ensure ensures the client that the operation is correctly finalized is waiting for matching replies from *all replicas.*

Response Quorum when Using Status Information and Audit Information.

Fortunately, integrating BFT forensics (Sheng et al. 2021) in the leader change sub-protocol (view change in PBFT or synchronization phase in BFT-SMaRt) enables the use of smaller response quorums, because we can leverage additional information obtained from the audit to consolidate the decision log.

More specifically, any equivocation that leads to correct replicas diverging into two quorums, respectively deciding o and o' and to additionally force a committed value to be erased, the at least $t_{fast}+1$ equivocators must participate in all three colored quorums (voting for o, o', and participating in the leader change) as outlined in Figure 6.7.

Assume that such a scenario happens, then the new leader runs the forensics protocol during the leader change, and identifies $t_{fast}+1$ equivocators (called *detections*). This audit information can be utilized to detect and ignore the status contributions of the equivocating replicas and instead wait for more status information from $t_{fast}+1$ more correct replicas (through the audit we know there must be more messages from slow but correct replicas in-flight). This approach is shown in Figure 6.7.

When utilizing status information from replicas and a PoC from the audit to determine the consistent outcome of the decision log, then instead of assuming

$$C > C' + t$$

we obtain

$$C + (t_{fast} + 1) > C' + t - (t_{fast} + 1) \tag{6.1}$$

By developing this equation (see the formulas below), we find that by *waiting for $n - t_{fast} - 1$ matching replies in fast mode, a client knows the result of its operation is finalized, ensuring the durability and linearizability of the replicated service* (see also Theorem 1 in Section 6.6).

Derivation of the Response Quorum $n - t_{fast} - 1$.

In this derivation, we assume for simplicity of exposition that $n = 3t + 1$. But the result works (and can be generalized) for any n.

$$n = 2t + C + C' \tag{6.2}$$

Inserting $n = 3t + 1$ into the equation :

$$3t + 1 = 2t + C + C' \tag{6.3}$$

Solving this equation for C':

$$C' = t + 1 - C \tag{6.4}$$

Further, remember that we require the inequality condition that

$$C + (t_{fast} + 1) > C' + t - (t_{fast} + 1) \tag{6.5}$$

Inserting 6.4 into 6.5:

$$C + (t_{fast} + 1) > t + 1 - C + t - (t_{fast} + 1) \tag{6.6}$$

$$2C > 2t + 1 - 2(t_{fast} + 1) \tag{6.7}$$

$$C > \frac{2t + 1 - 2(t_{fast} + 1)}{2} \tag{6.8}$$

$$C > \frac{2t - 2t_{fast} - 1}{2} \tag{6.9}$$

Now, we can calculate the response quorum Q for the client:

$$Q = 2t + C \tag{6.10}$$

which implies that

$$Q > 2t + \frac{2t - 2t_{fast} - 1}{2} \tag{6.11}$$

$$Q > \frac{6t - 2t_{fast} - 1}{2} \tag{6.12}$$

and the smallest integer solving Q is $3t - t_{fast}$ (which is the same as $n - t_{fast} - 1$). □

6.4 Implementation and Optimizations

The FlashConsensus prototype is implemented on top of AWARE (see Section 5.5) which is based on BFT-SMaRt. It is noteworthy, that the mechanisms employed in AWARE to measure latency and re-assign replica weights can be adopted to other quorum-based BFT SMR protocols. This implementation uses TLS to secure all communication channels and the elliptic curve digital signature algorithm (ECDSA) and SHA256 for signatures and hashes, respectively.

The modifications towards AWARE relate mostly to the switching mechanics between two modes of operations and the integration of the lightweight BFT protocol forensics procedure. A difference to the AWARE implementation (and WHEAT/BFT-SMaRt) is that the write messages and response messages to clients are signed by replicas using their ECDSA scheme (accept messages are already signed since the standard implementation of BFT-SMaRt to create externally verifiable proofs).

6.4.1 Implementation based on AWARE

Figure 6.5 presents the consolidated algorithm for FlashConsensus. This transformation summarizes all the modifications that are needed to enrich AWARE with the necessary mechanisms described in the previous section to obtain FlashConsensus.

The Client

The client invokes a new request by broadcasting a $\langle \text{REQUEST}, o \rangle$ message to all replicas, then waits to collect responses. Moreover, the client accepts the result to o depending on which mode the system operates. In the *fast* mode, the client needs to receive $n - (t_{fast} + 1)$ matching responses to ensure linearizability as derived in Section 6.3.3. When in *conservative* mode, obtaining a weighted quorum (weighted for the larger resilience threshold t) is sufficient. The client observes a mode switch as it would observe any modification of the systems's *view* object: After the mode changes on the replica-side, replicas increment the *view id*. As soon as replicas receive a client request with an outdated view number, they serialize the newest view object into the response message and transmit it back to the client, so that the client can update and synchronize its view object to match the replicas.

Timeout handling. When a request runs in a timeout, the client keeps re-transmitting the request and inspects the decision log of the replicas to check whether the request was committed. If the client receives too many non-matching results, it broadcasts a $\langle \text{PANIC}, o, rep \rangle$ message to the replicas. *rep* is the set containing all the responses that have been collected so far. Since responses are signed, a correct replica can verify that there are too many deviating outputs and request a synchronization phase, which will lead the system back to the conservative mode.

The Replica

The replica is based on the AWARE implementation which can be configured to run with the two different resilience thresholds. Before reconfiguration, AWARE stops the execution of consensus instances at a specific instance number. The reconfiguration changes the variable t in the view object and, then re-applies the weight calculations of the WHEAT scheme. After the view object is updated, the next consensus instance can be started. BFT-SMaRt employs a view store object, so that view objects (which define quorum systems) for all previous regimes are saved to the replica state. This means a proof for a certain consensus instance can always be correctly verified using the matching view.

Request handling. When a client request is received, each replica starts a request timer for it. The AWARE leader draws pending requests from its *toOrder* set (a set that contains all requests that have not yet been ordered), creates a batch of requests, and then proposes the batch using the fast regime or the conservative regime, depending on the system's current *mode* (which is part of the replica state).

Running forensics. The lightweight forensics procedure is started by a replica if it suspects that the fast mode is in danger of failing. This can be due to the following reasons:

- a client submitted a PANIC request with more than t_{fast} deviating responses.
- when exchanging the periodic checkpoint hashes, no stable checkpoint could be obtained.
- a synchronization phase was triggered (for instance because of a request timer triggered, a valid PoC received, or consensus latency disappointment)

A forensic procedure produces either a PoC, or $\varnothing$. In the case of $\varnothing$, the system is consistent. This might be the case if the forensic procedure is started because the total order multicast timeout (on a request) expires. A benign reason for this is a lack of synchrony in the system before GST. If a PoC is produced, it is employed in the synchronization phase to help the new leader decide on the final outcome of the decision log.

Periodic checkpoints and switching. At periodic intervals configured by θ of successful consensus instances, AWARE builds stable checkpoints and, if in conservative mode, attempts switching to the fast mode of operation. The optimization interval θ is inherited from AWARE and uses by default the same interval as the checkpointing interval.

6.4.2 Client-Side Speculation with Correctables

Operating in the fast mode necessitates clients to gather a significantly larger number of matching replies, specifically $n - t_{fast} - 1$, in order to maintain linearizability. This quantity far exceeds the usual requirement of $t+1$ replies in traditional BFT SMR protocols. When we further consider the geographic distribution of replicas across the world, it can be expected that collecting such a larger number of replies from replicas located in far-away regions may increase the latency experienced by clients.

FlashConsensus has the capability to further reduce the latency observed by clients through the utilization of client-side speculation. To achieve this, FlashConsensus integrates the concept of *correctables* (Guerraoui, Pavlovic, et al. 2016) inside the client interfaces of the BFT SMR algorithm. A *correctable* serves as a programming abstraction on the client side, that provides methods to a developer to access incremental consistency guarantees. This way, application performance can be enhanced by enabling speculative processing of intermediate results. Note that the state of a correctable can undergo multiple updates as long as responses are collected by the client, progressively strengthening the consistency guarantee until the correctable reaches its *final* state, representing the highest level of consistency. The design principles of the *Byzantine correctable* in FlashConsensus enable two goals:

Level	**Response quorum**	**Consistency**
first	1 response	none
weak	$Q_v = t_{fast} \cdot v_{max} + 1$	sequential consistency under t_{fast}
strong	$Q_v = 2t_{fast} \cdot v_{max} + 1$	linearizability under t_{fast}
final	$n - t_{fast} - 1$ responses	linearizability under t

Table 6.2: Incremental consistency levels are used for the *correctable* interface which a client application can access.

1. The correctable concludes in a *final* state which is equivalent to the standard BFT SMR safety guarantee (see Section 2.3.1), linearizability, under optimal protocol resilience $t = \lfloor \frac{n-1}{3} \rfloor$.
2. All consistency trade-offs that are less stringent may either relax resilience to non-optimal assumptions or make a trade-off, substituting linearizability for a less strict consistency model.

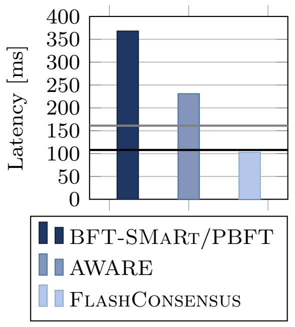
Table 6.2 summarizes how FlashConsensus defines incremental consistency levels. Note that given that BFT SMR shall preserve linearizability, only the *Strong* and *Final* levels provide standard SMR safety for their respectively assumed resilience.

- *First*: When a client receives its first response from any replica, it can immediately access the result and use it for speculation. This result may be incorrect, but the client logic can define how to deal with such a situation after correct results become available when the correctable is updated.
- A slightly better guarantee is provided by *Weak*, which demands replies from replicas accumulating $Q_v = t_{fast} \cdot v_{max} + 1$ weights, where v_{max} is the maximum weight of a replica, implying the result must have been confirmed by at least one correct replica if $f \leq t_{fast}$. This outcome may potentially be outdated, thus offering only sequential consistency (Attiya and Welch 1994) (not linearizability) and under the assumption that $f \leq t_{fast}$ failures holds.
- In the *Strong* state, a more stringent requirement of responses matching $Q_v = 2t_{fast} \cdot v_{max} + 1$ collected weights is imposed, thus ensuring linearizability under the same assumption of no more than t_{fast} failures. Note that this quorum ensures intersection in at least $t_{fast} + 1$ replicas between any read and write quorum, which is a necessary condition to preserve linearizability when read-only queries are implemented.
- Lastly, the *Final* level guarantees linearizability under t failures, just like any conventional BFT SMR protocol that includes the read-only optimization, by awaiting $n - t_{fast} - 1$ replies, as explained in Section 6.3.3.

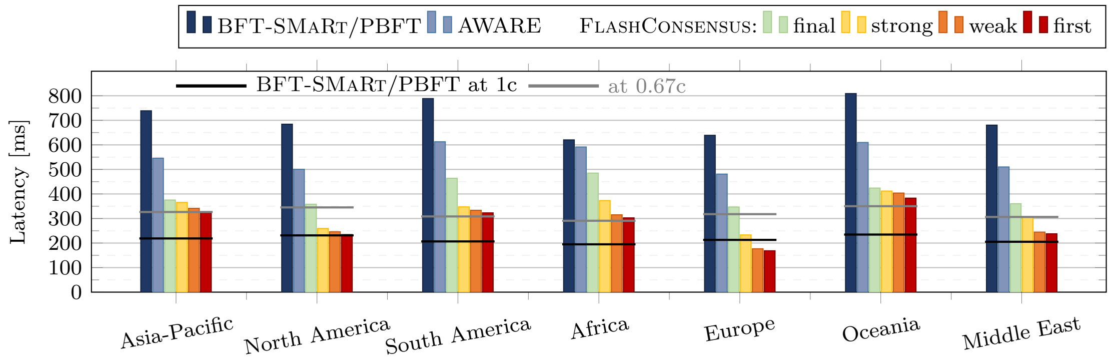
6.5 Evaluation

Through this section, we employ the implemented prototype of FlashConsensus to compare its latency with AWARE and BFT-SMaRt on the AWS infrastructure, as well as a simulated network containing 51 unique replica locations, where simulated latencies are based on real latency statistics from the Internet.

We conducted portions of our experiments within a local data center, employing cutting-edge, high-fidelity tools for network emulation and simulation, as shown in prior research (Gouveia et al. 2020; Jansen et al. 2022). Our primary research objective centers around latency measurements of the BFT SMR protocols. Latency in a planetary-scale system is fundamentally constrained by network link quality and the protocol's quorum formation rules in wide-area networks.



(a) Latency of consensus.



(b) Clients' measured request latency in protocol executions of BFT-SMaRt, AWARE, and FlashConsensus. The client measurements are averaged over all AWS regions per superregion.

Figure 6.8: Achieved speedups per superregion in the AWS cloud using $n = 21$ replicas spread across different regions.

In practical wide-area network (WAN) settings, we assess FlashConsensus against two baselines: BFT-SMaRt (a modern BFT replication protocol very similar to PBFT) and AWARE.

BFT-SMaRt/PBFT stands as the benchmark BFT SMR protocol due to its optimum communication step count and resilience threshold. PBFT represents a standard BFT SMR protocol.

AWARE, on the other hand, builds upon BFT-SMaRt by introducing self-tuning weighted replication, resulting in notable latency enhancements for WANs, as substantiated in previous studies (Sousa and Bessani 2015; Berger, Reiser, Sousa, et al. 2022). It exemplifies the cutting-edge approach to weighted BFT replication in WANs.

6.5.1 AWS Network

We start by exploring possible performance gains of FlashConsensusby contrasting measurements to both AWARE and BFT-SMaRt as baseline protocols. After that, we study how the induction of events (in particular, simulated failures) in a FlashConsensus's execution impacts the protocol latency and is handled by the inbuilt adaptiveness of FlashConsensus, responding with certain actions to the observed changes.

Setup

In this experiment, we employ a planetary-scale environment consisting of $n = 21$ AWS regions as shown in Figure 6.1a. In each region, we deploy two `c5.xlarge` instances, one carrying a replica and the other one carrying a client application. Each client sends 400-bytes requests simultaneously with all other clients. A client measures *request latency* as the observed time between sending a request and reaching a specific correctable state in FlashConsensus is reached or until BFT-SMaRt or AWARE can accept the result. Clients send the requests continuously to the replicas until each client has finished its measurement sample (which consists of 2000 requests per client). Note that a client request that arrives at the leader may experience a small waiting time until it is scheduled in the next consensus instance if there is currently an ongoing instance that needs to be completed first.

Moreover, clients are synchronous and only submit a single request at a time. This means, in their execution, they first await a result to a single outstanding request or a timeout event, and subsequently send their next request after randomly waiting for up to 1s. Finally, we employ the metric *request latency* as the average end-to-end protocol latency computed by a client after finalizing all operations. In FlashConsensus, we let the correctable compute a distinct *request latency* for every consistency level being reached.

Speedup Achieved through FlashConsensus

Figure 6.8 displays the results of all 21 clients clustered by the superregion they are placed in (this is to improve exposition of results) thus reporting their overregional averages.

The first observation that we make is that FlashConsensus significantly decreases consensus latency, leading to a speedup of 3.57× for reaching decisions (see Figure 6.8a). To put these results in perspective, the FlashConsensus latency (100ms) is more than 4× faster than the results reported for a recent optimistic protocol in a similar network (see Table 1 in (Lu et al. 2022)). Further, this is also better than the speedup of 2.29×, achievable if the speed of the network links used by BFT-SMaRt/PBFT approaches the speed of light[2]. The second observation we make is, that faster consensus decisions also lead to faster request latencies observed by clients distributed across the planet (see Figure 6.8b).

FlashConsensus leads to a speedup of 1.87× over BFT-SMaRt for clients' observed request latency with *Final* consistency (AWARE leads to 1.33× only) when averaging over all regions.

Moreover, FlashConsensus can yield even higher speedups by utilizing the incremental consistency levels of the correctable. For instance, the *Strong* consistency level, which provides linearizability as long as $f < t_{fast}$, accelerates request latency by 2.38×, and the speculative consistency levels *Weak* and *First* achieve speedups of 2.76× and 2.90×, respectively (the results are averaged over all regions).

Effect of Induced Failures during Execution

In our second experiment, we study the runtime behavior of FlashConsensus when there are faulty replicas in the system. For this purpose, we create an emulation of the AWS network in our local cluster, which resembles the real AWS network we used in the last subsection. This emulation allows us to be more flexible with the induction of events. The emulated network is constructed through the Kollaps network emulator (Gouveia et al. 2020), feeding it with latency statistics obtained from *cloudping.co*,[3] to mimic realistic latencies between AWS regions. Note that Kollaps has been used and validated for authentic WAN experimentation with BFT-SMaRt and WHEAT by (Gouveia et al. 2020).

We start FlashConsensus within this emulated network to explore its runtime behavior, particularly its reactions when inducing events. It is noteworthy that the observed request latencies show high variations. This is caused by the random waiting time of a request after arriving at the leader before it is batched and ordered (by starting the next consensus instance after the ongoing completed). Furthermore, we induce a predefined set of events, that we explain next to evaluate

[2]Notice that, in practice, it is accepted that the maximum speed internet links can reach is 0.67c (Cangialosi et al. 2015; Kohls and Diaz 2022).

[3]`https://www.cloudping.co/grid` is a website for monitoring AWS latencies.

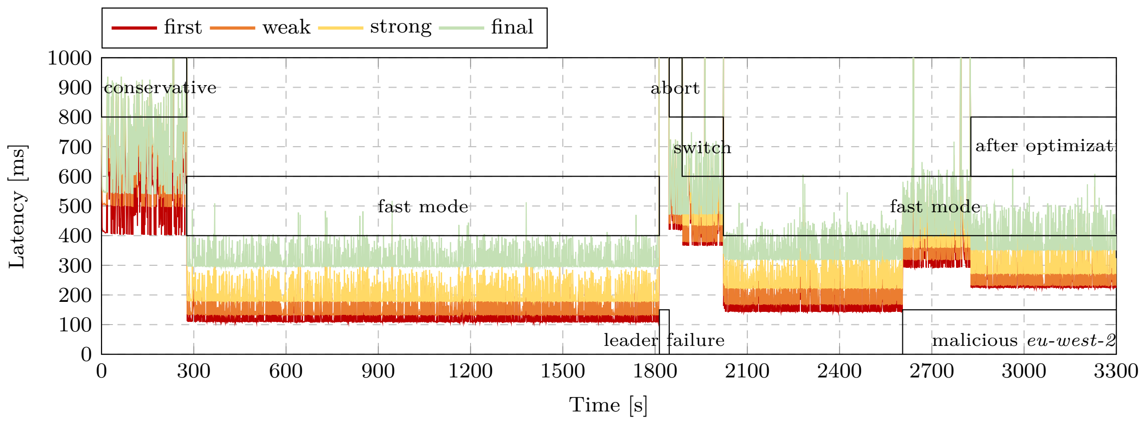


Figure 6.9: Runtime behavior of FlashConsensus under induced failures.

FlashConsensus's reactions. Figure 6.9 plots the measured request latency that is observed by a representative correct client.

Configuration switch. Initially, FlashConsensus launches in its conservative mode, in which it displays low performance. Subsequently, the system switches to the operational mode `fast` (around time 277 ¨s) leading to a significant decrease in latency when running with the optimized quorums. The system periodically reexamines whether it is running in the optimal configuration every 400 consensus instances (and may attempt switches at these intervals).

Silent leader. At the time 1814 s, the current leader becomes faulty by halting to participate in the protocol and instead remaining silent. This could be either seen as a malicious attempt to impede progress in the system or the effect of a common benign crash failure.

Following this, the replicas execute a leader change and abort the fast operation at the 1846-second mark, reverting to the conservative mode. The delay incurred during this phase is akin to what a client might encounter in the event of an equivocation detection.

By the 1889-second mark, after completing an additional 400 consensus instances, the replicas switch again to the fast mode. However, this mode transition only results in a modest performance enhancement since the weight optimization procedure has not yet been executed.

Finally, at the time 2022 s, the system undergoes self-optimization to achieve its optimal weight distribution and leader placement, using the inbuilt mechanisms of AWARE. Consequently, the system regains the low-latency performance it displayed previously.

Malicious leader. At the time 2640 s, we deliberately subjected the current leader to perform a *pre-prepare delay attack* (Amir, Coan, et al. 2010), wherein the Byzantine leader intentionally delayed sending its proposal and vote messages, leading to a degradation in the system's performance. This delay was implemented by manipulating the leader's behavior using the *tc* command.

Approximately 185 seconds later, which corresponds to the time required for a measurement round and self-optimization through the AWARE method, FlashConsensus detected that it was operating under a sub-optimal configuration. In response, it initiated changes to the replicas' weight distribution and leader location, thus re-locating the leader from `eu-west-2` to `us-east-1`, with the aim of speeding up the consensus.

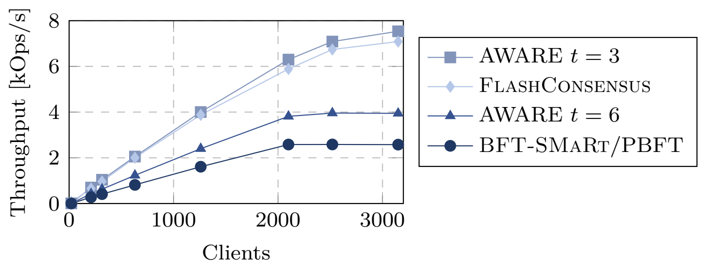


Figure 6.10: Throughput comparison for the AWS $n = 21$ replicas environment.

Throughput

While FlashConsensus primarily focuses on latency optimization, in this experiment we conducted a simple 0/0-microbenchmark within the emulated AWS network to reason about its throughput. This benchmark scales the workload by varying the number of clients that are launched across all AWS regions (spread uniformly), with the goal of measuring the throughput of four protocol (configurations): BFT-SMaRt, AWARE ($t = 6$), AWARE ($t = 3$), and FlashConsensus.

For the experiment, 0-byte requests and responses are used to prevent the saturation of link bandwidth. The outcomes of this experiment are illustrated in Figure 6.10, which display two key observations. First, higher throughput can be achieved through faster consensus decisions while the network bandwidth is not fully utilized. In particular, in its `fast` mode of operation, FlashConsensus achieves 2.5× the throughput of BFT-SMaRt and 1.8× the throughput of AWARE while maintaining the same degree of resilience. Second, FlashConsensus exhibits a minor performance drawback when compared to AWARE ($t = 3$), which employs the same quorums but offers only half of the resilience of FlashConsensus. We attribute this difference to the additional overhead incurred by FlashConsensus due to the integration of BFT forensic support, which necessitates the creation and dissemination of additional EDCSA signatures.

6.5.2 Larger Network

In our next experiment, we aim to ascertain whether the performance improvements observed in FlashConsensus remain consistent in an alternative scenario involving a greater number of replicas, specifically $n = 51$. This configuration approximates a typical deployment in a permissioned blockchain network. Given that this count surpasses the number of accessible AWS regions, we selected locations from a publicly accessible dataset generously provided by *Wonderproxy*[4].

Distribution. Using the wonderproxy dataset of latency statistics, we create a large network graph that covers 51 replica locations. Figure 6.11 shows how the 51 replicas and 12 clients are dispersed across the planet. The locations of the clients are chosen in a way to reflect observations made from different system sites.

Setup

We utilize *Phantom* (Jansen et al. 2022) as a state-of-the-art tool for simulating this network. Phantom uses a hybrid simulation-emulation architecture, in which real and unmodified applications directly run as native Linux processes which are plugged into a high-performance network simulation. We show in Chapter 7 how Phantom can be used to faithfully evaluate the performance of BFT protocol implementations and model realistic network environments.

[4]The interested reader can access the open dataset through the following URL: https://wonderproxy.com/blog/a-day-in-the-life-of-the-internet/.

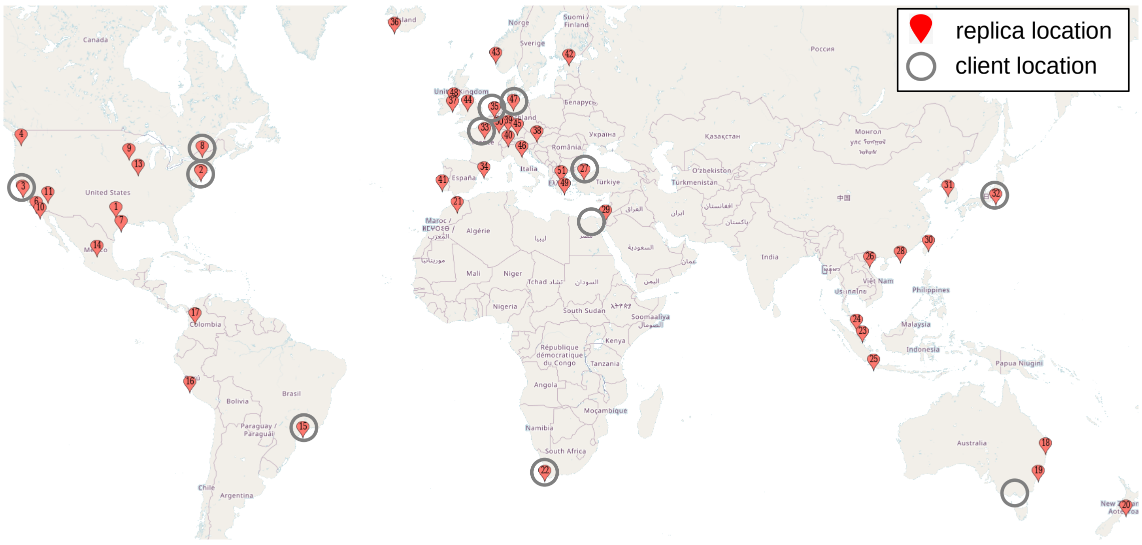


Figure 6.11: Map showing the locations of the 51 replicas.

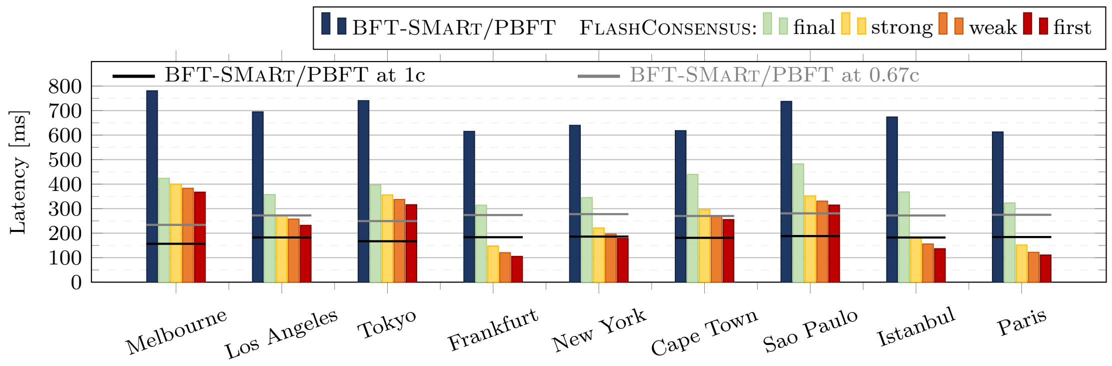


Figure 6.12: Latencies of BFT-SMaRt and FlashConsensus for $n = 51$ replicas, observed from different client locations.

In this experiment, we measure the speedup that FlashConsensus achieves and contrast it with BFT-SMaRt as a baseline. Like in the last subsection, we measure both consensus latency and client request latency using the same methodology as described for the AWS network (see Setcion 6.5.1). Additionally, for FlashConsensus, we distinguish the latencies necessary to reach every consistency guarantee of the correctable that the client manages.

Phantom starts replicas and clients in their virtual host locations with the initial protocol leader being deployed in *Cape Town*. Like before, clients run simultaneously and send requests with a payload of 400 bytes. Moreover, clients employ a randomized waiting interval of up to 4 before they submit the next operation.

It is noteworthy, that when Phantom bootstraps FlashConsensus, the system starts in its conservative configuration with resilience threshold $t = 16$, subsequently FlashConsensus optimizes this threshold to $t_{fast} = 8$, which happens before clients start collecting their measurement data (i.e., there is a warm-up phase before the benchmark is executed).

Results

Figure 6.12 displays latency enhancements comparable to those observed in the preceding experiment when comparing FlashConsensus with BFT-SMaRt. In the case of consensus latency, the reduction is quite significant, plummeting from 350 ms in BFT-SMaRt to a mere 88 ms in FlashConsensus. This translates to a consensus execution acceleration of 3.98×.

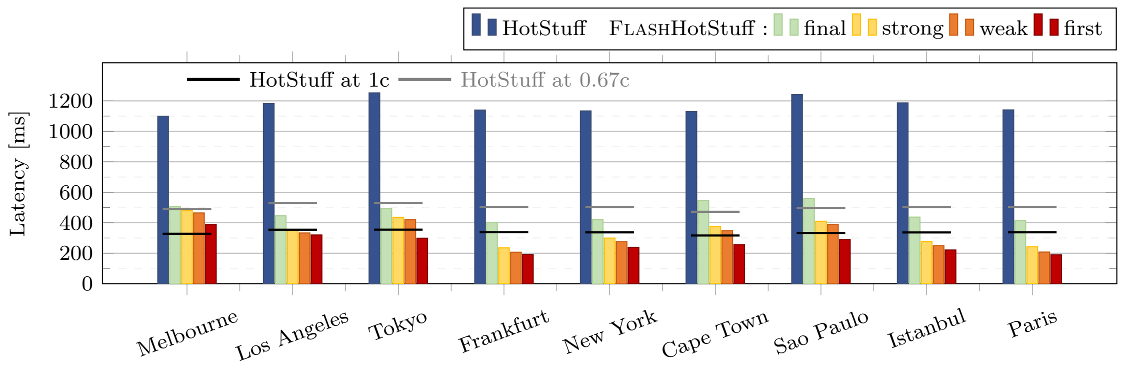


Figure 6.13: Latencies of HotStuff using FlashConsensus techniques for $n = 51$ replicas.

For request latencies involving *Final* consistency, the speedup varies among regions. For instance, *Frankfurt* experiences the most substantial improvement, with a speedup of 1.95×, lowering the latency from 615 ms to 314 ms. In contrast, *Cape Town* witnesses a more moderate enhancement, with a speedup of 1.41×, reducing the request latency from 618 ms to 440 ms.

Furthermore, the speedup becomes more pronounced when employing the incremental consistency levels offered by the correctable. To illustrate, in *Paris*, the *First* level attains an impressive speedup of 5.52×, while even at the *Strong* level, a notable speedup of 4.03× is maintained.

The average speedup over all client locations from BFT-SMaRt to FlashConsensus' *Final* consistency is 1.83× and becomes incrementally higher for the speculative levels *Strong* (2.70×), *Weak* (2.99×) and *First* (3.23×).

To put these numbers into context, if network links were nearly as fast as the speed of light, BFT-SMaRt would achieve a speedup of approximately 2.5×. These outcomes underscore the significance of employing smaller, weighted quorums as a critical strategy to attain low latency results that rival the expected latency of an optimal protocol (BFT-SMaRt/PBFT) under the ideal conditions of light-speed links.

6.5.3 FlashConsensus-flavored HotStuff

In principle, the ideas introduced by FlashConsensus are generic enough to be applied to several other quorum-based BFT SMR protocols in order to speed up the ordering of requests. For instance, the techniques of FlashConsensus could be used in multi-leader protocols such as MirBFT (Stathakopoulou et al. 2022) given that the best-connected replicas are the ones to be nominated as leaders, or even in BFT SMR protocols that provide additional guarantees such as fair ordering (Zhang et al. 2020). Yet, a critical question that must be clarified first is: *Does the BFT protocol implementation have forensic support?* Answering this question is non-trivial, since the availability of forensic support for a BFT protocol depends on very concrete implementation details: For instance, in PBFT, it is necessary that all protocol messages are signed (MACs are insufficient), while in HotStuff, it is necessary that the newView messages includes the view number of the *highQC* (the highest *prepareQC* a replica knows of), and that threshold signatures[5] are replaced by multi-signatures (Sheng et al. 2021). Provided forensic support is available, we can introduce a variant with FlashConsensus techniques for a BFT protocol.

In this subsection, to illustrate this versatility experimentally, we have applied the FlashConsensus transformation to HotStuff (Yin et al. 2019) (which can be enriched with strong forensic support (Sheng et al. 2021)) to provide a demonstration that latency reductions are also substantial in other BFT SMR protocol designs.

[5]Note that the original implementation of HotStuff, `libhotstuff`, does not implement threshold signatures, see `https://github.com/hot-stuff/libhotstuff`

HotStuff and FlashHotStuff. In contrast to both BFT-SMaRt and PBFT, the HotStuff protocol adopts an agreement pattern with an additional phase, enabling it to attain a linear communication complexity. In HotStuff, the leader collects and disseminates quorum certificates during each phase. As a consequence, all-to-all broadcast phases like in PBFT and BFT-SMaRt are no longer needed. It also means that HotStuff requires 7 communication steps[6] per agreement instance, in contrast to the 3 steps needed by BFT-SMaRt or PBFT, as illustrated in Figure 2.6. It is noteworhty, that sreamlined BFT protocols exist that leverage trusted execution environments to reduce this number of communication steps and these hybrid prococols operate faster in a WAN than HotStuff (Decouchant, Kozhaya, et al. 2022; Decouchant, Kozhaya, et al. 2024).

The HotStuff design renders the overall system's latency particularly sensitive to the speed at which agreement can be reached, which in turn depends on the pace at which a HotStuff leader can succeed in collecting and distributing its quorum certificates which depends on quorum formation rules in a WAN.

FlashHotStuff strategically selects a system configuration in which the leader establishes rapid communication with a set of well-connected replicas endowed with high voting weights. Further, we suppose that FlashHotStuff can successfully operate in its tentative fast mode, in which the lower resilience threshold t_{fast} holds.

Setup

In this experiment, we employ a prediction model of the HotStuff protocol. This model is an implementation of the HotStuff protocol pattern but uses weighted quorums to create quorum certificates. The overall model is thus similar to the one AWARE uses to anticipate the effect of weight and leader changes during its self-optimization (see Section 5.3.2). We simulate FlashHotStuff and the normal (non-weighted) HotStuff protocol runs on the wonderproxy latency map using $n = 51$ replicas (see Figure 6.11) thus using the same latency map as in the previous experiment. In these simulations, we calculate the attainable latencies for various consistency levels within FlashHotStuff, as well as the latency of the original HotStuff. Additionally, we determine the hypothetical latencies of HotStuff assuming network link speeds approximate the speed of light.

Results

Simulation results clearly demonstrate that FlashHotStuff yields a significant enhancement in latency compared to the original HotStuff protocol, as illustrated in Figure 6.13.

In terms of consensus latency, FlashHotStuff optimizes the performance remarkably, reducing it from 853 ms in HotStuff to just 177 ms. This translates to a consensus execution speedup of 4.82×, achieved through the integration of weighted replication, strategic leader placement, and the utilization of smaller quorums.

For client request latencies, specifically with *Final* consistency, the speedup varies across different locations, with the highest observed in *Frankfurt* (2.84×) and the lowest in *Cape Town* (2.07×).

Consistent with previous observations, the speedup becomes more pronounced with lower consistency levels. For instance, in *Paris*, the *First* level achieves a substantial speedup of 6.03×, reducing latency from 1141 ms to 189 ms. Even at the *Strong* consistency level, a noteworthy speedup of 4.71× is achieved, lowering latency to 242 ms.

On average across all client locations, transitioning from HotStuff to FlashHotStuff at the *Final* level results in a speedup of 2.56× (noteworthily while maintaining the same consistency guarantee, namely linearizability). This speedup improves incrementally when employing the speculative

[6]HotStuff requires 7 communication steps when counting the communcation steps from PREPARE to DECIDE but it would be actually 8 when also counting an additional NEW-VIEW message before PREPARE.

consistency levels provided by the correctable: *Strong* ($3.69\times$), *Weak* ($4.05\times$), and *First* ($4.74\times$), respectively.

6.6 Correctness of FlashConsensus

In the following, we prove the correctness of FlashConsensus transformation, as summarized in Figure 6.5. In particular, we prove that it preserves the safety and liveness of its underlying protocol, AWARE, with up to t failures.

To abstract the exact service being implemented on top of FlashConsensus, our proofs consider the replicated decision log abstraction as described in §2.3, in which every operation needs to be allocated consistently in a given position/slot of the log.

6.6.1 Safety

Instead of only proving all operations are executed in total order, we have to prove all clients observe operations in total order, which ensures linearizability (Herlihy and Wing 1990). To prove that, we start by showing that an operation finalized in some position of the decision log in a mode of operation is kept in that position after a mode switch.

Proposition 6.6.1

Let o be an operation finalized in conservative *mode in the i-th position of the decision log. After the system switches to subsequent* fast *mode, o will still be the i-th operation executed in the system.*

Proof. Assume, without loss of generality, that the system switches from *conservative* to *fast* mode right after executing its j-th operation, with $j \geq i$. In this case, all operations ordered after the switch will be finalized in positions after j (Rule S4 of Figure 6.5), and thus the effect of previously ordered operations (including o) will be maintained. □

Lemma 6.6.1

Let o be an operation finalized in fast *mode in the i-th position of the decision log. After a synchronization phase that switches the system to the* conservative *mode, o will still be the i-th operation executed in the system.*

Proof. Let us assume, for the sake of contradiction, that an operation o finalized in the i-th position of the decision log in fast mode does not appear in this position after a synchronization phase. This can happen for one of two reasons:

1. *Operation o was erased, making the i-th position empty.* This can only happen if there is no correct replica in the intersection between the quorum of $n - t_{fast} - 1$ replicas that informed the client about the finalization of o and the quorum of $n - t$ replicas that informed the new leader about the requests ordered in fast mode during the synchronization phase. This is not possible since these quorums intersect in $(n - t_{fast} - 1) + (n - t) - n = \frac{3t}{2}$ replicas[7], which is clearly bigger than t, for $t \geq 2$ (when the fast mode brings benefits).
2. *Operation o was replaced by another operation o' in the i-th position.* This happens only if the number of replicas reporting o' as prepared is higher than the number of replicas reporting o in the synchronization phase quorum of $n - t$ replicas (Rule S6 step 2 of the transformation in Figure 6.5). Since o was finalized, at least $C = n - t_{fast} - 1 - t$ *correct replicas* informed the client about the execution of o. Under an equivocation scenario, operation o' would be

[7]This value is obtained by using $t_{fast} = \frac{t}{2}$ and $n = 3t + 1$.

reported by at most $C' = n - C + t = t + t_{fast} + 1$ replicas, including the equivocators that participated in the decision of o and o'. Notice that the $t_{fast}+1$ replicas that maliciously voted in the preparation of the two requests can be detected by the lightweight forensics protocol used in the synchronization phase (Rule S6 step 1 of the transformation). Therefore, these replicas will be ignored in the synchronization phase quorum (Rule S6 step 2) (and later expelled from the system—Rule S6 steps 3 and 4), ensuring no more than t replicas report o' in this quorum. Consequently, even in this worst-case scenario, the new leader will always see $n - 2t > t$ replicas reporting o, being thus the most common value appearing in the reported values. This result will make o be kept in the i-th position of the log after a synchronization phase.

□

The following theorem proves FlashConsensus' main safety property: finalized operations are observed in the same position of the decision log by every correct client.

Theorem 6.6.1

If an operation o is finalized in i-th position of the decision log, then no client observes an operation $o' \neq o$ in this position of the decision log.

Proof. We start by observing that, according to Rule C2 of the transformation (Figure 6.5), a client accepts an operation result as finalized only if the number of matching REPLY messages *in the same mode* satisfies certain quorum sizes. Let $mode(o)$ be the mode of operation in which o was finalized, and assume, for the sake of contradiction, that an operation $o' \neq o$ appears as finalized in the i-th position for some correct client.

To show correct clients cannot observe different operations o and o' in i-th position of the system history, we need to consider all possible combinations of modes for o and o'. We start by considering the cases in which both operations were finalized in the same mode without any switch in the system mode between their executions. There are two cases to consider:

1. $mode(o) = mode(o') = conservative$: if both operations were executed in the conservative mode, then the total order of operations ensured by AWARE's state machine replication algorithm (Berger, Reiser, Sousa, et al. 2022; Sousa and Bessani 2012) makes it impossible for different clients to observe finalized operations o and o' in the same position in the system history.

2. $mode(o) = mode(o') = fast$: if both operations were ordered in fast mode, then $f > t_{fast}$ malicious replicas (including the leader) can lead to different correct replicas deciding different operations for the same position i of the decision log. In this case, we have to show that clients waiting for $n - t_{fast} - 1$ matching replies is enough to ensure they cannot observe two finalized operations in the same position. This holds because the size of the intersection of any two reply quorums is $(n - t_{fast} - 1) + (n - t_{fast} - 1) - n = n - t - 2$, which is bigger than t for any $t \geq 2$ (when the fast mode can be used). This means correct clients will not observe two finalized operations in the same decision log position in our system model.

Now, we need to consider the cases in which there was exactly one mode switch between the finalization of o and o'. There are two cases to consider:

1. $mode(o) = conservative$ and $mode(o') = fast$: Proposition 6.6.1 states that finalized operation o position cannot be altered in a conservative-to-fast switch, i.e., it will still appear as the i-th operation executed, and thus operation o' cannot appear in position i.

2. $mode(o) = fast$ and $mode(o') = conservative$: Lemma 6.6.1 states that if o was finalized in fast mode, a subsequent synchronization phase that switches the system to conservative mode keeps the o in the i-th position of the decision log, making thus impossible for another operation o' to be finalized in this position.

Lastly, we need to consider all the cases above, but in which multiple mode switches happen between the finalization of o and o'. Proposition 6.6.1 and Lemma 6.6.1 ensure a finalized operation is preserved in the system state even after a mode switch. Consequently, by applying induction arguments, it is easy to see that the number of switches between the finalization of o and o' does not change the fact o will always remain the i-th operation in the decision log. □

6.6.2 Liveness

As defined in §2.3, SMR liveness comprises the guarantee that operations issued by correct clients are eventually finalized. Proving the liveness of BFT protocols is generally perceived as considerably more intricate than proving their safety (Bravo et al. 2022b). However, this distinction does not apply to FlashConsensus because our transformation relies upon the underlying protocol (AWARE, a variant of the well-known BFT-SMaRt) for ensuring this property. The following theorem defines the liveness of FlashConsensus.

> **Theorem 6.6.2**
>
> *An operation issued by a correct client will eventually be finalized.*

Proof. To argue about that, suppose a correct client c sends operation o to the replicas. As stated before, an operation is finalized if the client observes it was executed in a certain position by sufficiently many replicas in the same mode (Rule C2 of Figure 6.5). Again, we have to consider the two modes of FlashConsensus:

1. *Conservative mode:* If the leader is correct and there is sufficient synchrony, then a batch containing o will be eventually decided through AWARE (Rule S1). In case of a faulty leader or asynchrony, the timers associated with o on correct replicas will expire, and the synchronization phase will be executed until a correct leader forces replicas to decide a batch containing o (potentially after GST). At this point, at least $n - t$ correct replicas will send matching replies, which are collected by c until it has a weighted response quorum (Sousa and Bessani 2015) to finalize o.

2. *Fast mode:* In fast mode, before GST no liveness can be ensured, and the system will switch back to the conservative mode. After GST, we have to consider two cases:

   a) Case $f \leq t_{fast}$ and correct leader: This case is analogous to the conservative mode because AWARE$\star$ solves consensus with up to t_{fast} faulty replicas.

   b) Case $t_{fast} < f \leq t$ or faulty leader: In this case, AWARE$\star$ *might not* be finished, and the timers associated with o (Rule S1) will expire in correct replicas. This will cause replicas to initiate the synchronization phase (Rule S5), switching to the conservative mode, in which the request will be ordered and finalized, as explained above. Alternatively, f Byzantine replicas might participate in consensus but not reply to c, which will never be able to collect $n - t_{fast} - 1$ matching replies. In this case, the client periodically reattempts to confirm the result of o by checking the log of decisions (Rule C2). This can be repeated until the next periodic checkpoint is formed to verify in which position on the decision log o appears (Rule S3). This will eventually happen because if o is not ordered successfully, a timeout will trigger at the replicas, causing them to initiate the synchronization phase and switch the protocol to the conservative mode. The liveness argument from the conservative mode then eventually holds for o.

□

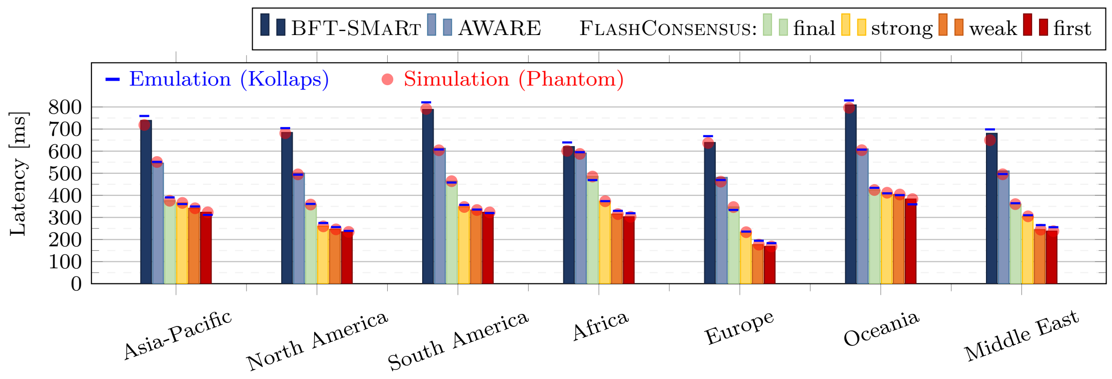


Figure 6.14: Comparison of clients' observed end-to-end latencies for protocol runs with BFT-SMaRt, AWARE and FlashConsensus in different network environments: real, emulated, and simulated. The client results are averaged over all regions per continent.

6.7 Validation of Emulated/Simulated Networks

In this section, we compare the results of the experiment conducted in Section 6.5.1 for which we used the real AWS cloud infrastructure with supplementary experiments that use an emulated and simulated network environment that mimic the AWS infrastructure by using latency statistics from cloudping. For these network environments, we use state-of-the-art network emulation and simulation tools, namely Kollaps (Gouveia et al. 2020) and Phantom (Jansen et al. 2022). The repeated experiments should give some insights into how close real network characteristics can be modeled using emulation and simulation tools. A threat to validity is that the network statistics average latency observations over a larger period (i.e., a year). In contrast, when conducting experimental runs, short-time fluctuations might make individual network links appear faster or slower than usual, which can impact the speed at which certain quorums are formed. Nevertheless, we are convinced that using the network tools and latency statistics creates reasonably realistic networks that can be used to validate the latency improvements of the quorum-based protocols we are working with.

In Figure 6.14, we contrast protocol runs for BFT-SMaRt, AWARE, and FlashConsensus on a real network, as well as in the Kollaps-based and Phantom-based networks. On average, we observed latencies that were 1.5% higher in the emulated network than on the real AWS network. Respectively, the simulated network yielded latency results that were, on average, 0.8% lower than the real network.

6.8 Additional Related Work

In this section, we contrast the approach of FlashConsensus with some related works. Some of these works, in particular, touching on adaptivity in BFT SMR systems as well as geographically-distributed SMR systems have been already mentioned in the discussion of related work for AWARE in Section 5.9 and remain related to FlashConsensus.

WHEAT and AWARE. WHEAT improves latency of BFT SMR by utilizing weighted replication and tentative executions (Sousa and Bessani 2015), while AWARE enhances WHEAT through self-monitoring capabilities and a dynamic optimization method which adjusts weights and leader position (Chapter 5). In contrast, FlashConsensus can accelerate Byzantine consensus even further than AWARE by utilizing BFT forensics to use smaller consensus quorums safely, thus solving the resilience-performance dilemma that originally affected AWARE and enabling considerable latency benefits.

Client-side speculation. PBFT-CS enriches PBFT with client-side speculation (Wester et al. 2009). In PBFT-CS, clients can speculatively send subsequent operations after predicting a response to an earlier, uncommitted operation. Yet, all clients need to track and propagate the dependencies between the operations. This is not a requirement in FlashConsensus, where the implemented correctable abstracts and simplifies client speculation, thus allowing an application developer to more easily program with well-defined consistency levels.

Geographically-distributed SMR. There is a bunch of research works that studies how to improve SMR systems in geographic deployments (Mao, F. P. Junqueira, et al. 2008; Mao, F. Junqueira, et al. 2009; Wester et al. 2009; Veronese et al. 2010; Amir, Danilov, et al. 2010; Coelho and Pedone 2018; Numakura et al. 2019; Eischer, Straßner, et al. 2020; Eischer and Distler 2020; Neiheiser, Rech, et al. 2020). A notable difference between them and FlashConsensus is that FlashConsensus assumes a strict BFT model, without relaxing assumptions on fault behavior or inter-region dependencies (i.e., assuming at least a single correct replica within the same system site like in (Mao, F. Junqueira, et al. 2009)) or a specific architecture of the system, i.e., deployment of replicas.

Adaptive Approaches in BFT SMR. Integrating the idea of adaptivity into BFT SMR protocols has been explored in several research works (Aublin, Mokhtar, et al. 2013; Bahsoun et al. 2015; S. Liu and Vukolić 2017; Eischer and Distler 2018; Carvalho et al. 2018; D. S. Silva et al. 2021; Chiba et al. 2022; Nischwitz et al. 2022; Lu et al. 2022). For a broad overview, we refer the reader to Section 3.2. A few notable examples include RBFT (Aublin, Mokhtar, et al. 2013) which monitors system performance under redundant leaders to prevent performance degradation through malicious leaders, optimizing the leader selection (e.g., (S. Liu and Vukolić 2017; Eischer and Distler 2018)), adaptively switching the consensus algorithm (Bahsoun et al. 2015; Carvalho et al. 2018; Lu et al. 2022), being network-agnostic (providing more resilience in synchronous networks) (Blum et al. 2020) or adapting the state transfer strategy to the available network bandwidth (Chiba et al. 2022). Bolt-Dumbo (Lu et al. 2022) runs a fast quorum-based protocol in synchronous periods and falls back to an asynchronous consensus otherwise. ThreatAdaptive (D. S. Silva et al. 2021) can adjust (i.e., increase) the employed fault threshold parameter t (i.e., by increasing the system size n) to counter an perceived increase in adversarial strength, thus making a BFT SMR system more resilient. In contrast to these ideas, FlashConsensus can be more seen as a transformation that is applicable to many quorum-based BFT protocols and accelerates planetary-scale Byzantine consensus by making both system configuration (leader and replica weights) *and* the resilience threshold adaptive without impacting resilience through the integration of BFT forensics.

Fast or speculative BFT SMR variants. Research has demonstrated that incorporating extra redundancy and employing a suboptimal resilience threshold can be effectively harnessed to create faster consensus variations, such as the two-step Byzantine consensus (Martin and Alvisi 2006; Howard et al. 2021; Kuznetsov et al. 2021), and one-step asynchronous Byzantine consensus when dealing with contention-free scenarios (Friedman et al. 2005; Song and Renesse 2008). We provide a more in-depth overview of fast variants in Section 3.3. In contrast to these variants, the idea behind FlashConsensus is to extend from a protocol like PBFT that is optimal in respect to the number of communications steps while achieving maximum resilience, and then to optimize latency *without making a compromise on resilience*. There are also speculative BFT protocols, a prominent example is Zyzzyva (Kotla et al. 2007). Evaluations on BFT systems found that, *speculative execution*, as utilized in Zyzzyva, demands a network environment that is both predictable and stable, which is uncommon in geo-distributed deployments, specifically, Zyzzyva's performance degrades primarily because it requires responses from *all* replicas within a *strictly configured time window* to complete a request in a single phase (Singh et al. 2008).

BFT protocol forensics. BFT forensics was introduced as a method for post-event analysis of safety breaches within BFT protocols, as described in (Sheng et al. 2021). This analysis can yield results indicating that, in the event of equivocation, at least $t+1$ culprits can be identified, with accountability extending to as many as $\frac{2n}{3}$ potentially Byzantine replicas. Polygraph (Civit, Gilbert, and Gramoli 2021), on the other hand, represents an accountable Byzantine consensus algorithm tailored specifically for blockchain applications and it enables the penalization of culprits, such as through stake slashing, when equivocations occur. Furthermore, a straightforward method for transforming any Byzantine consensus protocol into an accountable Byzantine consensus protocol has been presented in (Civit, Gilbert, Gramoli, et al. 2022). The Basilic protocol class offers a solution to the consensus problem, assuming up to $n \leq 3t+d+2q$ replicas while tolerating t general Byzantine failures, d deceitful failures (those violating safety and detectable through forensics), and q benign failures, as explained in (Ranchal-Pedrosa and Gramoli 2023). It is worth noting that Basilic's resilience has been proven as optimal. In addition, IA-CCF, as presented in (Shamis et al. 2022), demonstrates that logging all message exchanges within PBFT on a blockchain allows for the identification of any misbehaving replicas in case of equivocations. In contrast, FlashConsensus does not distinguish between various types of failures or necessitate the use of a blockchain. Instead, it employs a simplified version of the BFT forensics protocol outlined in (Sheng et al. 2021) to pinpoint a restricted number of equivocators within a fast regency (or *view*, in the context of PBFT). FlashConsensus enables the safe, tentative utilization of smaller quorums in the system and is potentially applicable to many quorum-based BFT systems.

6.9 Concluding Remarks

In this chapter, we explained FlashConsensus, a transformation for quorum-based BFT SMR protocols. We showed how to derive FlashConsensus from the underlying base protocol AWARE (see Chapter 5) by integrating BFT forensics techniques and using it for a novel application of primarily as a safeguard against equivocations *during runtime*.

Summary and key insight. The key insight of FlashConsensus is that planetary-scale Byzantine consensus can be expedited by fusing adaptive weighted replication with BFT forensics to safely let the BFT SMR protocol (in our exemplary case: AWARE) run in a *fast operation mode* in which smaller quorums can be accessed to steer consensus decisions. We found that the tentative use of this fast mode does not force a compromise in resilience. Instead, FlashConsensus can always guarantee both linearizability (as the safety property) as well as liveness under optimal resilience ($n = 3t+1$). This is because when optimal resilience is actually needed, i.e., in a situation in which the actual number of faulty replicas is $f = \lfloor (n-1)/3 \rfloor$, then the auditing capability of FlashConsensus lets the system quickly recover and abort to a *conservative mode* in which the optimal ($n = 3t+1$)-resilience guarantee is met.

Observations obtained from a planetary-scale study. Our evaluation reveals significant latency advantages of FlashConsensus, with a remarkable speedup of 1.87× when compared to BFT-SMaRt. Our approach, which is transferrable to other BFT protocols, has the potential to enhance their performance, especially in scenarios involving geographic dispersion. Furthermore, we have demonstrated that protocols that utilize more communication steps in their agreement patterns, such as HotStuff, can benefit even more from the transformation, i.e., achieving a 2.56× speedup. Additionally, we recognize the potential of client-side speculation, where the application can selectively opt for a relaxed consistency level according to the client-replica contract at the granularity of individual operations. This approach may be particularly suitable for time-sensitive and low-value transactions, like micro-payments, as these can experience significant speedups of up to 6×.

Chapter 7
Simulation of BFT SMR

Modern BFT SMR protocols are placing growing emphasis on the scalability aspect to fulfill the demands of distributed ledger technology, such as achieving decentralization and geographical dispersion. Thoroughly assessing the performance of scalable BFT protocol implementations demands careful and extensive evaluation, possibly considering different scenarios for the envisioned practical deployment. Evaluations can help to ascertain that performance goals are met, and, to possibly identify settings in which the system may present a bottleneck, e.g., in terms of network bandwidth or message dissemination time.

While experiments with real protocol deployments usually offer the best realism, they are costly and time-consuming. In this chapter, we propose an alternative (and cost-effective) approach to large-scale cloud deployments, which involves utilizing *network simulations* to predict the performance of BFT protocols while the environment can be scaled experimentally.

A key novelty in our simulation method is that existing BFT protocol *implementations* are plugged into the simulation without necessitating any code modifications or re-implementations, as we consider these to be both time-consuming and error-prone. In this chapter, we initially describe our simulation architecture, which facilitates scalable performance evaluation of unmodified BFT systems through the utilization of high-performance network simulations.

After that, we assess the accuracy of network simulations in predicting the performance of BFT systems by comparing simulation outcomes to real measurements obtained from systems deployed on cloud infrastructures. Our findings indicate that simulation results offer a reasonably close approximation, particularly when dealing with larger system scales, as the network ultimately becomes the primary limiting factor affecting system performance.

In the last part of this chapter, we employ the developed simulation approach to assess the performance of PBFT and BFT protocols that belong to the "blockchain generation", such as the popular HotStuff protocol (Yin et al. 2019) and Kauri (Neiheiser, Matos, et al. 2021) in realistic, planetary-scale environments, and, when faults are induced.

This chapter summarizes the main insights that were partly published in two previous research articles:

> Christian Berger, Sadok Ben Toumia, and Hans P. Reiser (2022). "Does My BFT Protocol Implementation Scale?" In: The 3rd International Workshop on Distributed Infrastructure for Common Good (DICG). Quebec, Quebec City, Canada: Association for Computing Machinery, pp. 19–24. ISBN: 9781450399289. DOI: 10.1145/3565383.3566109. URL: https://doi.org/10.1145/3565383.3566109

> Christian Berger, Sadok Ben Toumia, and Hans P. Reiser (2023). "Scalable Performance Evaluation of Byzantine Fault-Tolerant Systems Using Network Simulation". In: *The 28th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE

7.1 Motivation and Overview

Over the past few years, there has been a notable trend in the blockchain world. It involves the consideration of “classical” BFT protocols, as inspired by PBFT (Castro and Liskov 1999), as potential replacements for the energy-hungry Proof-of-Work (Nakamoto 2008) mechanism. This shift aims to adopt a more efficient approach for achieving consensus among all valid blockchain replicas when determining which block should be added next to the ledger (Vukolić 2015).

While traditional BFT protocols such as PBFT can achieve this objective, the cost of running agreement among a large number of replicas results in a sharp performance decline in large-scale systems. As a result, there has been a quest for potential solutions aimed at enhancing the scalability of BFT protocols, giving rise to the emergence of numerous new BFT protocol designs (Yin et al. 2019; Crain et al. 2021; Cason et al. 2021; Neiheiser, Matos, et al. 2021; Stathakopoulou et al. 2022; P. Li, G. Wang, Chen, Long, et al. 2020).

Deployments on cloud infrastructures. To demonstrate the ability of these innovative BFT protocols to deliver adequate performance in real-world, large-scale systems, it is necessary to thoroughly assess their operational behavior.

To achieve this, research papers proposing these protocols typically include an evaluation section involving large-scale deployments. These evaluations use experimental setups that are carried out on cloud infrastructures such as AWS, where experiments involve deploying numerous nodes, often reaching several hundred (as exemplified in (Yin et al. 2019; P. Li, G. Wang, Chen, Long, et al. 2020; Cason et al. 2021; Crain et al. 2021; Neiheiser, Matos, et al. 2021), among others). These deployments serve to showcase the performance of a BFT protocol on a large scale.

While evaluation involving cloud-based protocol deployments typically provides the highest level of realism, it makes experimentation both expensive and time-consuming, especially when testing multiple protocol configurations. Hence, a compelling option for affordable and swift validation of BFT protocol implementations, potentially while they are still in the development phase, is to conduct simulations to forecast system performance.

Simulating BFT protocols. BFTSim (Singh et al. 2008) was the first simulator designed for an “apples-to-apples” evaluation of BFT protocols, but it falls short in terms of scalability, rendering it less suitable for the modern “blockchain generation” of BFT protocols. Evidently, it can only effectively simulate up to $n = 32$ PBFT replicas (P.-L. Wang et al. 2022). Additionally, BFTSim necessitates the modeling of a BFT protocol using the P2 language (Loo et al. 2005), a task that can be somewhat error-prone, given the complexity of BFT protocols, and time-consuming.

A more contemporary tool, as introduced in (P.-L. Wang et al. 2022), offers the capability for scalable simulation of BFT protocols. However, it comes with the drawback of necessitating a comprehensive re-implementation of the BFT protocol using JavaScript. Additionally, this tool lacks the capability to make predictions regarding system throughput, which consequently restricts its performance evaluation to the observation of latency.

In our approach, we bridge a critical gap in the current state-of-the-art by introducing a simulation methodology for forecasting BFT system performance that is highly scalable, without necessitating any re-implementation of the protocol. This key feature distinguishes our approach from existing methods and represents a significant advancement for predicting BFT system performance in a practical way.

Why not just use network emulation? Emulation aims to replicate the precise behavior of the entity being emulated. Emulators such as Kollaps can replicate experiments conducted on AWS with BFT protocols on a local server farm (Gouveia et al. 2020). One notable advantage of emulation is its ability to maintain a high degree of realism, as BFT protocols continue to function in

Implementation	BFT SMR Protocol	Language	Repo on github.com
HotStuff-secp256k1	HotStuff	C++	/hot-stuff/libhotstuff
Themis (PBFT)	PBFT	Rust	/ibr-ds/themis
BFT-SMaRt	BFT-SMaRt	Java	/bft-smart/library
HotStuff-bls	HotStuff (bls-signatures)	C++	/Raycoms/Kauri-Public/
Kauri	Kauri	C++	/Raycoms/Kauri-Public/

Table 7.1: BFT protocol implementations that we simulated.

real-time and interact with real kernel and network protocols. However, unlike simulation, emulation is not as resource-efficient, as it runs the real application code in real-time, thus demanding a substantial number of physical machines for conducting large-scale experiments.

In contrast, simulation disentangles the concept of simulated time from real-time and uses abstractions to expedite executions. The simulation procedure involves capturing relevant aspects through a model, essentially meaning that the simulation only mimics (and to some extent simplifies) the environment of the protocol, or potentially also the actual protocol behavior if the application model is re-implemented.

The simulation approach offers the benefits of simplified experimental control, superior reproducibility (ensuring deterministic protocol executions), and enhanced scalability in comparison to emulation. However, a potential drawback that remains is the question of *validity of results*, as the model may not entirely and accurately mirror real-world conditions.

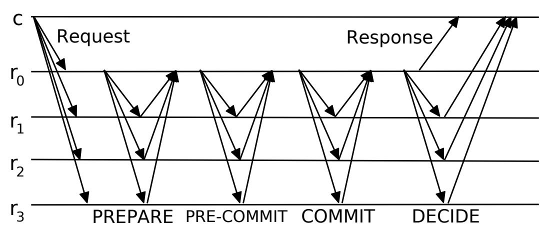
A network approach to simulating BFT protocols. In our approach, we initially aim to address a prevailing limitation shared by all existing BFT simulators: the ability to seamlessly integrate an *existing*, *unmodified* BFT protocol implementation into a high-performance and scalable network simulation, without necessitating any re-implementation or modifications to the source code. By adopting this approach, we can ensure the validity on the application level, as we employ the real application binaries to initiate real Linux processes, subsequently connecting them to the simulation engine.

However, it's important to note that a potential threat to the validity of our approach lies in the fact that when we rely solely on network simulation, it may overlook the impact of processing time (i.e., considering CPU-intensive tasks) on performance within BFT protocols, such as the generation and verification of signatures.

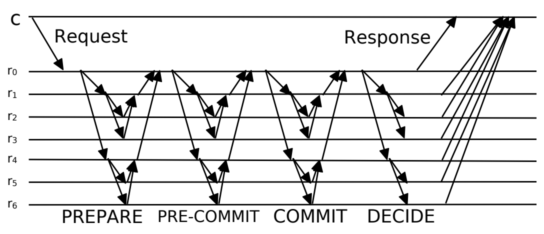
Interestingly, our discoveries revealed that simulation-based performance results can offer a valuable approximation of real measurements, especially in large-scale systems. This phenomenon occurs because, once a certain system size of replicas is reached (typically at $n \geq 32$), the overall system performance is primarily dictated by the underlying network, which consistently acts as a performance bottleneck. We provide more detailed insights on this in our validity analysis in Section 7.3.5.

7.2 BFT Protocols under Test

To showcase the effectiveness of our approach, we apply it to the well-known PBFT protocol (Castro and Liskov 1999) as well as BFT protocols belonging to the "blockchain generation", specifically HotStuff (Yin et al. 2019) and Kauri (Neiheiser, Matos, et al. 2021). To improve scalability over PBFT, these protocols innovate through more scalable communication patterns. To acquaint the reader with the different *communication strategies* employed by these protocols, we we offer a brief overview in the following and also provide some details on the used implementations (see Tabel 7.1).



(a) In **HotStuff**, the leader collects the votes from the other replicas and distributes quorum certificates to all other replicas.



(b) In **Kauri**, the leader uses a balanced communication tree for aggregation and dissemination of quorum certificates.

Figure 7.1: Communication patterns of novel BFT protocols (star and tree).

7.2.1 PBFT and BFT-SMaRt

PBFT (Castro and Liskov 1999) encounters scalability issues when it comes to larger system sizes because all operations are channeled through a single leader. This leader is responsible for distributing blocks (batches in PBFT parlance) to all other replicas, causing the leader's up-link bandwidth to act as a bottleneck, impeding the overall system's performance.

Additionally, PBFT's practice of all-to-all broadcasts results in a communication strategy with quadratic costs, i.e., leading to the transmission of $O(n^2)$ messages (along with their authenticators) within the system. We refer the reader for a more detailed overview of PBFT to Section 2.3.2. BFT-SMaRt (Bessani, Sousa, et al. 2014) uses the same communication strategy for its normal case operation as PBFT does, thus the same reasoning applies to BFT-SMaRt as well (a comprehensive overview of BFT-SMaRt is detailed in Section 2.3.3).

Implementations. For the reason of compatibility with modern operating systems, we employ a Rust-based re-implementation (see (Rüsch et al. 2019b)) of PBFT and not the original implementation by Miguel Castro. Further, we use the official Java-based implementation of BFT-SMaRt (Bessani, Sousa, et al. 2014) that is still being maintained actively on its github repository.

7.2.2 HotStuff and Kauri

HotStuff. HotStuff achieves a linear message complexity by letting the leader collect votes from all the other replicas and then distribute a quorum certificate, as explained in (Yin et al. 2019). To reduce the expense associated with transmitting message authenticators, the leader can employ an aggregation technique to compress a quorum of $n - t$ signatures into a single fixed-size threshold signature. As a result, the threshold signature has a constant size of $O(1)$, which represents a significant enhancement compared to the transmission of individual signatures, which would yield the size $O(n)$. While HotStuff adds one more phase compared to PBFT, it can also maintain linear communication for its view change (which has quadratic costs in PBFT). As illustrated in Figure 7.1a, the communication pattern remains at an imbalance, wherein each follower replica solely communicates with the leader, resembling a *star* topology. Meanwhile, the leader is still tasked with communicating with all the other replicas.

Kauri. Employing a communication topology based on a tree structure provides a distinct advantage since it spreads the duty of aggregating and disseminating votes and quorum certificates, thereby alleviating the leader's burden. Kauri, as detailed in (Neiheiser, Matos, et al. 2021), is an example of a tree-based BFT SMR protocol based on HotStuff (see Figure 7.1b). It introduces a timeout mechanism for aggregation to handle potential failures at the leaf nodes. To address potential failures of internal nodes within the tree structure, Kauri uses a reconfiguration scheme. This scheme ensures *fast* (meaning after at most $t + 1$ attempts) identification of a correct set of internal nodes, provided that the number of failures remains below a specific threshold. The increased latency resulting from the additional communication steps in the tree is alleviated through a more advanced pipelining approach. This approach is more sophisticated compared to the pipelining mechanism used in HotStuff, as it can initiate multiple agreement instances for each protocol stage, whereas HotStuff initiates only a single agreement instance per stage.

Implementations. The initial implementation of HotStuff, referred to as HotStuff-secp256k1, uses elliptic curves and does not include signature aggregation, which means it does not combine multiple signatures into a single fixed-size signature (Yin et al. 2019). Subsequently, an alternative implementation, provided by (Neiheiser, Matos, et al. 2021), integrated BLS signatures (Boneh et al. 2004) into HotStuff, which we denote as HotStuff-bls. This implementation is largely based on the former but provides the capability for signature aggregation. Moreover, the implementation of Kauri (Neiheiser, Matos, et al. 2021) additionally allows to use a balanced tree for communication with customizable fanout (by default $\lceil\sqrt{n}\rceil$) and a variable pipeline depth (which can be used to improve performance despite a higher number of communication steps).

7.3 Methodology

In this section, we explain the methodology we developed. First, we justify and motivate our goal of assessing the performance of real BFT protocol implementations through an extensive simulation of the running distributed system. Within this simulation, our method creates *replica* and *client* components as Linux processes through the designated BFT protocol implementations. All components are seamlessly integrated into an event-based simulation engine that is responsible for constructing and governing the system environment.

Following this, we present the software architecture of our simulation methodology. This approach entails the user submitting a basic experimental description file (EDF), specified in YAML format to a *frontend*. Subsequently, the frontend proceeds to generate all necessary runtime components, establish a realistic network topology, and schedules a new experiment. This scheduling process involves the initiation of a *backend* instance responsible for executing the network simulation. A comprehensive illustration of this architecture can be found in Figure 7.2.

Subsequently, we validate the simulation results by conducting a comparative analysis with measurements obtained from real-world deployments which we mimic by employing our simulation methodology.

BFT Protocol Selection

We substantiate our choice of BFT protocols as follows: Our primary objective was to illustrate how different communication strategies (such as all-to-all, star, and tree) affect system performance. Consequently, we opted for a single “representative” BFT protocol corresponding to each strategy — specifically, PBFT, HotStuff, and Kauri. In future work, we intend to expand our evaluations to include multi-leader and leaderless BFT protocols, which can also be evaluated using our methodology.

7.3.1 Why Simulate BFT Protocol Implementations?

There are several good reasons to explore BFT protocol implementations through the lens of our simulation method. We discuss a few of them in the following:

Plug & Play Utility

A crucial advantage of our specific approach lies in its novel *plug-and-play* functionality. This ensures the realism of executed BFT protocols since we directly employ the real protocol implementation to launch genuine Linux processes that serve us as the application model, thereby replicating real implementation behavior. More specifically, this approach can be regarded as emulation at the application level. Simultaneously, users are spared from the need for re-implementation or extensive modeling efforts, which can easily introduce errors given the intricate nature of BFT protocols. Such re-implementation (or modeling) attempts can also become a time-intensive task.

Integration Tests

Moreover, the simulation of real BFT protocol *implementations*, as opposed to custom-crafted models, proves valuable for expedited prototyping and validation. Notably, certain bugs at the implementation level in BFT systems may not manifest in typical scenarios with a common setting of $n = 4$ replicas. Hence, simulations offer a means to perform comprehensive *integration testing* on a larger scale. Furthermore, developers can employ these simulations for automated regression testing, assessing whether their recent commits have had adverse effects on protocol performance for large-scale deployments. For instance, we found a problem in the view change mechanism of the employed Themis (PBFT) implementation that only manifests at $n \geq 32$ (see Figure 7.13c).

Controlled Experimentation

There are several inherent advantages associated with the use of simulations. To begin, simulations provide a cost-effective means of studying the runtime dynamics of BFT protocols, substantially more affordable than deploying them in real-world scenarios. Presently, there is a growing trend toward open-source publication of BFT protocol implementations. Simulations allow comparisons of these protocols under standardized conditions, enabling equitable assessments of their performance in situations where network performance becomes a critical factor. In particular, simulations offer the capability to methodically explore the parameter space encompassing both protocol variables and network conditions within a controlled environment, yielding deterministic results.

Realistic Network Environments

Specific to our approach is that we generate *extensive network topologies* by employing large latency maps. This is possible, because we use statistical data (i.e., a large data set) that is available from Wonderproxy[1]. This enables us to encompass a broader range of regions than what is typically available through most cloud providers, such as Amazon's AWS infrastructure.

Didactical Purposes

Last but not least, simulations can also serve an educational role. They can contribute to a deeper comprehension of the behavior of BFT protocols in large-scale deployments. For instance, this method can be employed to enhance the teaching experience in universities' distributed systems labs, allowing students to acquire practical insights and hands-on experience by working with the many BFT protocols that have been already implemented.

7.3.2 Accelerating Large-Scale Simulations of BFT Protocol Implementation

Even when assuming that a suitable simulation engine (the "backend", which we explain afterward) is available, running large-scale simulations of BFT protocols necessitates approaching a variety of challenges first:

1. **Network topologies must be realistic**. The quality of simulation depends on the use of realistic and expansive network topologies suitable for varying system sizes. Ideally, the

[1]For additional information on these statistics, we refer the interested reader to https://wonderproxy.com/blog/a-day-in-the-life-of-the-internet/.

communication links within these topologies should closely mirror those found in real-world deployments. This is essential to ensure realistic simulations of wide-area network environments.

2. **Bootstrapping BFT protocols needs tooling support**. Assistance is required when configuring the deployment of BFT protocol implementations. Bootstrapping BFT protocol implementations in a plug-and-play fashion involves a multitude of steps that can be both laborious and prone to errors, particularly because they are protocol-specific. Examples of these steps include generating protocol-specific runtime artifacts like cryptographic key materials or configuration files, which vary for each BFT protocol.
3. **Exploration of BFT protocol configuration space is difficult**. When developing and testing BFT algorithms, often numerous experiments are conducted due to the different combinations of protocol settings. Since simulations operate in virtual time, these experiments can take hours, depending on the host system's specifications. For the sake of user convenience and a seamless experience, it is important that experiments can be specified in batches and run consecutively without requiring manual user intervention.
4. **Supervising the experimentation**. Monitoring and evaluating resource consumption during simulation runs, such as CPU utilization and memory usage, may be necessary.

For these reasons, we undertook the development of a *frontend* tool[2], which operates in conjunction with the Phantom simulator (Jansen et al. 2022). This tool aims to streamline and accelerate the evaluation of unmodified BFT protocol implementations.

7.3.3 Frontend Design and Experimental Description

The *frontend* comprises multiple components (as depicted in Figure 7.2) and adheres to a modular architecture. It is designed to be adaptable rather than being customized for a particular BFT protocol, thus making it easily extensible.

Scheduler

The toolchain is overseen by a scheduler responsible for the coordination of all tasks. This includes environment preparation, configuration of runtime artifacts for a BFT protocol, and the initiation of a resource monitor. The scheduler employs *protocol connectors* to establish a BFT protocol setup and loads *experiment description files* (refer to Figure 7.3 for an example specifying a single experiment). Experimental description files can contain a series of experiments designated for a chosen BFT protocol. Subsequently, once an experiment is prepared for execution, the scheduler launches the simulation engine, Phantom.

Additionally, the scheduler initializes a resource monitor to capture data related to resource utilization, such as memory allocation and CPU time, throughout the simulation runs. These statistics can serve as indicators for potential vertical scaling requirements of the host machine and as estimations for the resources needed to conduct extensive simulations.

Protocol Connector

To simulate each BFT protocol implementation, the creation of protocol configuration files and the generation of necessary cryptographic keys are essential tasks. Given that protocol configurations and cryptographic methods can differ significantly between individual BFT protocol implementations, we have designed a protocol-specific setup procedure encapsulated as a tool named the *protocol connector*. This connector is called by the scheduler initially, after parsing the EDF and passing the experiment specification.

[2]The code and experiment files are available as open-source on GitHub (`https://github.com/Delphi-BFT/tool`).

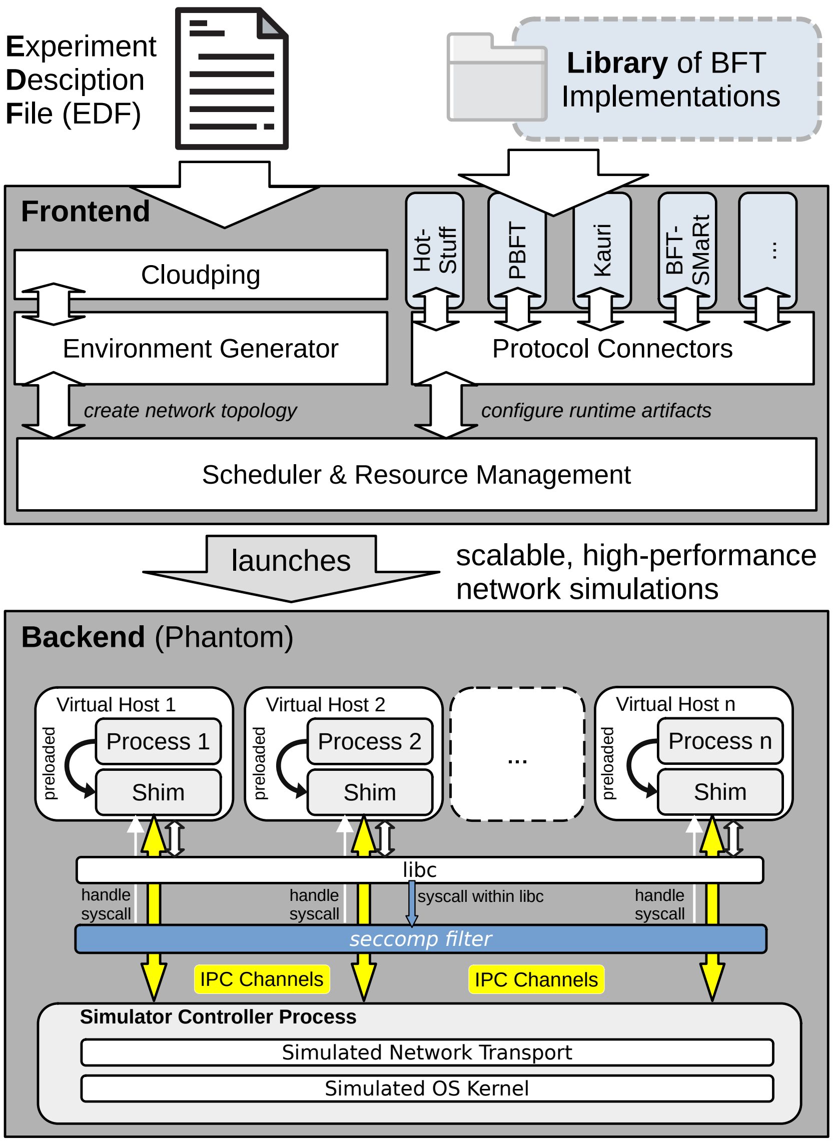


Figure 7.2: Architecture employed in our simulation method.

To ensure seamless integration of new BFT protocols into our toolchain, each connector must implement the `build()` and `configure()` methods. This approach simplifies the expansion of our toolset to integrate novel BFT protocols, as it merely entails the development of a new protocol connector. Based on our experience, this typically involves writing between 100 and 200 lines of code.

Environment Generator

The environment generator is responsible for constructing network topologies by creating complete graphs suitable for any system size. These network topologies closely mimic realistic deployment scenarios, whether in a LAN or WAN context. To generate network graphs that reflect a realistic WAN deployment of nodes with heterogeneous communication link latencies, the environment generator leverages a *cloudping* component. This component retrieves actual round-trip latency data between all AWS regions from Cloudping[3]. This enables the tool to construct network topologies that resemble realistic BFT protocol deployments on the AWS cloud infrastructure.

Additionally, we have implemented an expanded latency map incorporating 51 distinct locations,

[3]For reference, please visit https://www.cloudping.co/grid.

EDF
```
protocolName: hotstuff
protocolConnectorPath: ./connectors/hotstuff.js
```

EDF.network
```
bandwidthUp: 25 Mibits
bandwidthDown: 25 Mibits
latency:
map: 'aws21'
replicas: ['eu-west-1': 2, 'us-east-1': 1, 'sa-east-1': 1]
clients: ['ca-central-1': 2]
packetLoss: 0.0
```

EDF.replica
```
replicas: 4
blockSize: 400
replySize: 1024
timeout: 4000
```

EDF.client
```
clients: 16
numberOfHosts: 2
requestSize: 1024
outStandingPerClient: 175
```

EDF.faults
```
type: crash
threshold: 0.25
timestamp: 60 s
```

Figure 7.3: Example for an experimental description file (EDF).

which we derived using latency statistics from Wonderproxy. The cloudping component can either fetch an up-to-date latency map from an online source or utilize one from the existing repository. It is important to note that maintaining determinism and ensuring reproducibility necessitates the use of the same latency map. The `EDF.network` description file defines the distribution of replicas and clients across this latency map, along with the configuration of bandwidth and packet loss.

Fault Induction

The frontend also offers the capability to introduce faults during simulation runs, a critical feature for assessing system performance under adverse conditions. As BFT protocols may employ different resilience thresholds, we enable users to specify their desired threshold for inducing faults. We model this as a static threshold adversary, a common assumption in BFT protocols. One of the supported fault types is `type: crash`, which terminates the faulty replica processes at a designated `timestamp` within the simulation.

Another scenario we can simulate is a denial-of-service (DoS) attack. By setting `type: dos` and specifying an `overload` parameter, we instantiate a malicious client that sends a significantly larger number of requests (e.g., 100× the normal rate) to assess the system's ability to handle such an overload. This helps evaluate whether implementations can effectively limit the number of requests accepted from a single client and maintain a fair batching strategy.

Additionally, we support `packetloss`, which defines the packet drop ratio at the network level (though this setting currently requires configuration in the `EDF.network` section). This feature allows us to evaluate how well BFT implementations perform in scenarios where the network exhibits packet loss.

As a current limitation, more sophisticated Byzantine fault scenarios are not supported, but we see this topic as a currently open research direction. For an exploration of more complex fault scenarios, we may, i.e., seek inspiration from the Twins methodology (Bano, Sonnino, Chursin, et al. 2022). Twins is an unit test case generator designed to simulate Byzantine behavior through the replication of cryptographic IDs of replicas, thus producing protocol runs that feature equivocations or forgotten replica states. We plan to explore the integration of Twins within our simulation methodology in future work.

7.3.4 Using Phantom to Simulate BFT Protocols as Native Linux Processes

As mentioned before, running the simulation is performed by Phantom which acts as the *backend* in our method while the frontend converts the user inputted EDF to Phantom-specific files. In our workflow, the scheduler triggers the execution of Phantom as soon as a new simulation experiment is prepared and the host's hardware resources become available.

Phantom operates a hybrid simulation/emulation architecture: Applications run as regular processes on the Linux operating system and are seamlessly integrated into the simulation through a system call interface, utilizing standard kernel functionalities (Jansen et al. 2022). This approach offers the distinct advantage of preserving the realism of the application layer, as it facilitates the execution of real BFT protocol implementations. Phantom is also resource-friendly and runs on a single machine.

Through its hybrid architecture, Phantom employs a middle ground between two distinct paradigms: the pure simulator ns-3 (Riley and Henderson 2010) and the pure emulator Mininet (Lantz et al. 2010). Phantom strikes a balance between sufficient application realism necessary for BFT protocol execution and resource-efficiency (and scalability) compared to emulators. Thus Phantom seems promising for the purpose of BFT protocol research, which is why we chose it as the backend for conducting the simulations.

Simulated Environment

Phantom describes a network topology as a directed graph (called the *environment*). Within this graph *virtual hosts* are nodes and communication links between hosts are the edges. This graph gets attributed to adding the environmental details, e.g., virtual hosts (the nodes) can specify their available uplink and downlink bandwidth while communication links (the edges) can specify latency and packet loss.

Each virtual host may execute one or more *applications* (which fall into the category of being either *clients* or *replicas* for our purpose), resulting in the instantiation of real Linux processes. These Linux processes are started by the simulator's *controller process* and subsequently managed by a Phantom worker. The Phantom worker employs the use of `LD_PRELOAD` to preload a shared library referred to as the *shim* which enables the co-opting of its managed processes into the simulation environment (as illustrated in Figure 7.2). To enhance interception capabilities, a second method is employed that extends the `LD_PRELOAD` method by employing `seccomp` (secure computing). The use of a seccomp filter within a process permits the interception of system calls that cannot be preloaded. For more details on how these interception strategies work, we refer the interested reader to the Phantom paper (Jansen et al. 2022).

Simulation Engine

The *shim* establishes an inter-process communication channel (IPC) with the simulator controller process and intercepts functions at the system call interface. While the shim is capable of directly emulating certain system calls, the majority of system calls are relayed to and managed by the simulator controller process. This controller process is responsible for simulating kernel and networking functionalities, including aspects such as time progression, I/O operations related to `file`, `socket`, `pipe`, `timer`, event descriptors, and the transmission of packets.

Deterministic Execution

Phantom maintains determinism throughout the simulation process by using a pseudo-random generator. This generator is initialized with a seed derived from a configuration file, enabling the faithful emulation of all necessary sources of randomness during the simulation. This includes emulating functions like `getrandom` or reads from `/dev/*random`. Furthermore, each Phantom worker enforces single-threaded execution across all the processes it oversees. This design ensures that the remaining managed processes or threads remain in an idle state, effectively preventing concurrent access to the memory of these managed processes (Jansen et al. 2022). This yields the benefit of deterministic protocol runs. If a *Byzenbug* (Bessani, Sousa, et al. 2014) (a bug that only occurs in rare conditions, similar to *Heisenbug* (Gray 1986), but manifests only if it occurs in more than t replicas simultaneously) is detected, then the whole protocol run can be re-played for a reproduction of the bug – a huge advantage for testing.

7.3.5 Validation

In this subsection, we conduct a validity analysis. For this purpose, we contrast the measurements obtained from real BFT protocol executions with the results derived from our simulation experiments mimicking these real deployments.

HotStuff-secp256k1

We aim to replicate the evaluation setup outlined in the HotStuff paper (available in the arXiv version (Yin et al. 2018)) for the purpose of comparing their real-world measurements with our simulation outcomes.

Setup. In their setup, they deployed over a hundred virtual machines within an AWS data center. Each machine possessed a bandwidth of up to 1.2 GB/s, and the latency between any pair of machines was reported to be less than 1 ms (which we must treat as 1 ms in our simulation since we lack a better estimation).

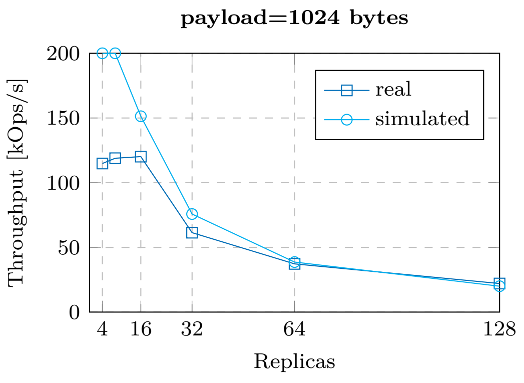
The block size used in their setup was 400. We undertake a comparative analysis against two series of measurements: "p1024" where the payload size for requests and responses is 1024 bytes, and "10ms" featuring empty payloads but with the latency of all communication links set to 10 ms.

Our goal is to investigate how faithfully the performance of HotStuff can be predicted by regarding only the networking capabilities of replicas, which manifests at the point where the network becomes the bottleneck for system performance.

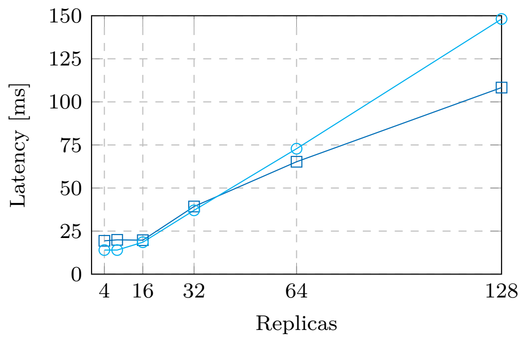
Results. We present our findings in Figure 7.4. The simulation results for the payload experiment ("p1024") exhibit a similar trend, with a decrease in throughput observed for $n \geq 32$, and approaching the real measurement series. For smaller replica groups (see $n = 4$ in Figure 7.4c), the network simulation predicts higher performance, achieving up to 200k op/s. This aligns with the theoretical maximum constrained only by the 1 ms link latency, allowing pipelined HotStuff to commit a block of 400 requests every 2 ms[4].

As the HotStuff performance becomes increasingly constrained by available bandwidth (particularly evident at $n \geq 32$), the disparity in throughput diminishes. Similarly, our results for the "10ms" setting closely align with the real measurements: 80 ms in the simulation compared to 84.1 ms in reality, and 20k op/s in the simulation versus 19.2k op/s in reality for $n = 4$. However, as

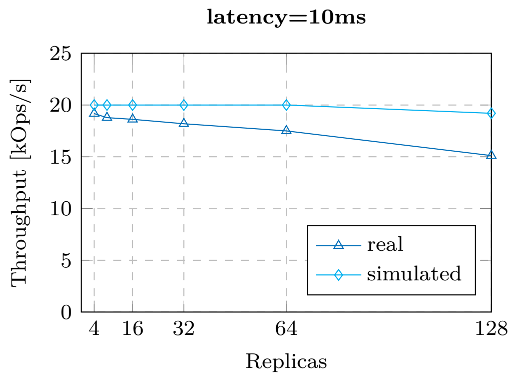
[4]Pipelined HotStuff can commit a block after each pipeline stage. Each HotStuff pipeline stage is a protocol phase and consists of two communication steps: The leader collecting a quorum certificate and then disseminating it. In this LAN experiment, HotStuff commits roughly 500 blocks per second and each contains 400 requests, resulting in 200k tps.



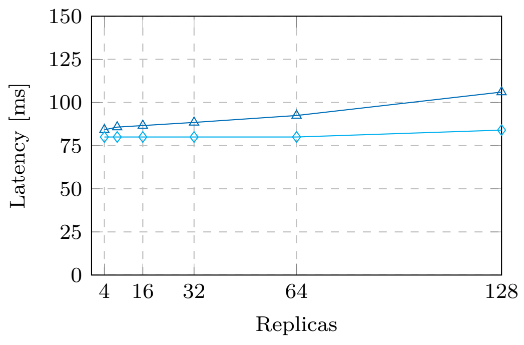
(a) Throughput.



(b) Latency.

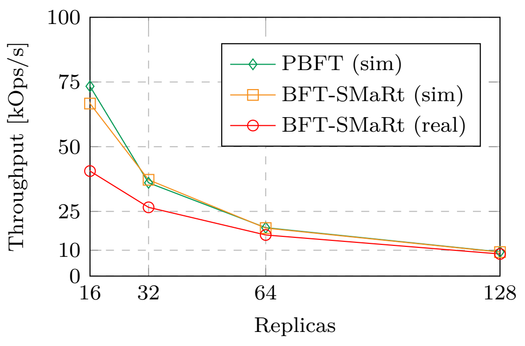


(c) Throughput.

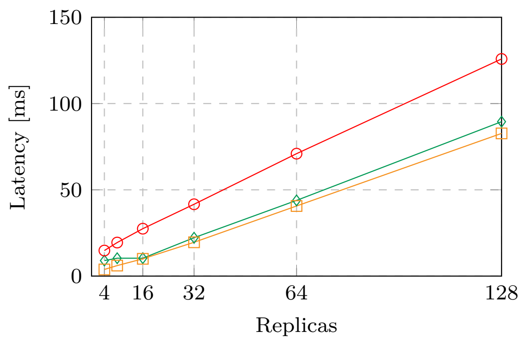


(d) Latency.

Figure 7.4: Comparison of performance results of HotStuff-secp256k1 vs. its simulated counterpart using a bandwidth of 10 Gbit.



(a) Throughput.

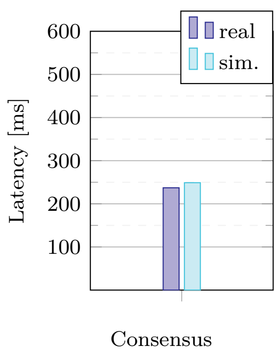


(b) Latency.

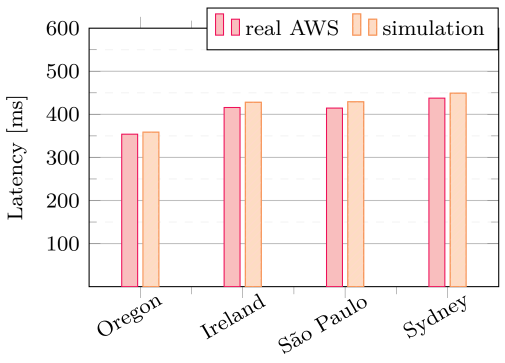
Figure 7.5: Performance of simulated BFT-SMaRt, simulated PBFT and a real BFT-SMaRt execution in a 10 Gbit LAN.

n increases, the difference grows[5], such as 84 ms vs. 106 ms and 19.2k op/s vs. 15.1k op/s for $n = 128$. It is worth noting that this "10ms" experiment, which employs requests without any payload, is less sensitive to network bottlenecks, which are typically induced by limited replica bandwidth when n becomes larger.

[5]The reason for this is that the "10ms" experiment uses requests with 0 byte payload, thus the CPU processing delays (i.e., for verifying an increasing number of signatures — these delays are not captured in the simulation) seem to become a more relevant factor than delays incurred by the limited network bandwidth of 10 Gbit/s.



(a) Consensus latency.



(b) End-to-end request latencies observed by clients in AWS regions.

Figure 7.6: Comparison of a real BFT-SMaRt WAN deployment on the AWS infrastructure with its simulated counterpart.

BFT-SMaRt and PBFT

In our subsequent experiment, we validate the performance of BFT-SMaRt and PBFT[6]. To achieve this, we leverage measurements obtained from (Yin et al. 2018) as well as conducting our own experiment on a wide-area network (WAN) constructed using four distinct AWS regions.

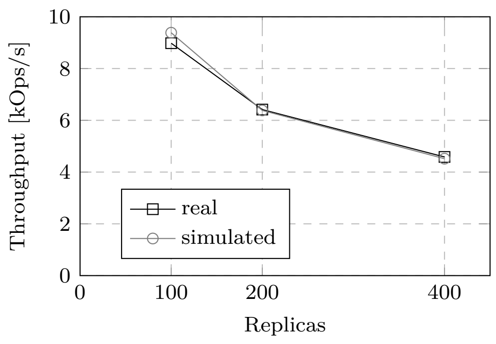
LAN Setup. Initially, we model the "p1024" setup outlined in (Yin et al. 2018), as it provides us real measurement data available for BFT-SMaRt. In this setup, we create a network environment with a 1 ms latency and a 10 Gbit bandwidth. We conduct simulations for both BFT-SMaRt and PBFT, since they employ an identical message pattern in the normal operation phase of both protocols.

Results. Our findings are presented in Figure 7.5. Notably, we observe that in the initial phase ($n \leq 32$), the real performance results for BFT-SMaRt fall notably below the predictions from our simulations. However, this dynamic shifts with an increase in n. For instance, at $n = 128$, we observe 9,280 op/s for BFT-SMaRt (in comparison to the real measurement of 8,557 op/s) and 9,354 op/s for the PBFT implementation. In the latency graph, we notice a distinct gap between the real and simulated BFT-SMaRt results. We believe the main reason for this difference lies in our inability to precisely replicate the request sending rate as conducted in (Yin et al. 2018), as it was not explicitly specified in their experimental setup.

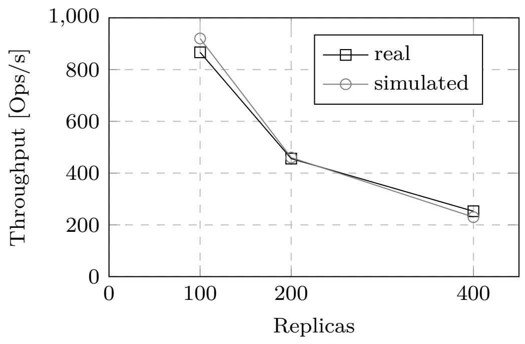
Geographic Dispersion. Subsequently, we conduct an experiment involving the geographical distribution of BFT-SMaRt replicas, where each replica is situated in a distinct AWS region. This experimental configuration aligns with studies that investigate latency enhancements, for instance, in (Sousa and Bessani 2015). For this purpose, we configure a scenario with $n = 4$ and designate the regions Oregon, Ireland, São Paulo, and Sydney for the deployment of a replica and a client application each. We conduct the experiment by sequentially running clients, with each client sending 1000 requests without payload. The end-to-end latency is recorded for each request, while the leader replica (based in Oregon) monitors the consensus latency of the system.

Results. We observe that the consensus latency is only marginally higher in the simulation (237 ms compared to 249 ms). Additionally, the simulation results exhibit slightly higher end-to-end request latencies across all clients (refer to Figure 7.6). The disparity between the simulated and

[6]For PBFT, we employ a Rust-based implementation available at `github.com/ibr-ds/themis` since the original implementation by Castro et al. (Castro and Liskov 1999) does not compile on modern computer operating systems.



(a) Kauri.



(b) HotStuff-bls.

Figure 7.7: Reproducing the "`global`" scenario from (Neiheiser, Matos, et al. 2021) that uses 100 ms inter-replica latency and 25 Mbit/s bandwitdh.

real executions is the lowest in Oregon, with a deviation of only 1.3%, and the highest in São Paulo, with a deviation of 3.5%.

In a similar experiment we compared this simulation method with an emulator (Kollaps) and the real AWS infrastructure using more replicas and regions ($n = 21$) for the protocols BFT-SMaRt, AWARE and FlashConsensus. For the results on this we refer the interested reader to Section 6.7.

Kauri and HotStuff-bls

In our next experiment, we validate the performance of Kauri and HotStuff-bls, which are new BFT protocol implementations that aim to improve performance over the original HotStuff implementation.

Setup. For this purpose, we mimic the "global" experiment from Kauri (Neiheiser, Matos, et al. 2021), which employs varying numbers of replicas (specifically, 100, 200, and 400 replicas). Kauri's "global" system setup assumes that these replicas are interconnected over a planetary-scale network, where each replica is allocated 25 Mbit/s of bandwidth and connected over a 100 ms latency link when communicating with any other replica. Our validation encompasses two implementations introduced by (Neiheiser, Matos, et al. 2021): HotStuff-bls, which is an adaptation of HotStuff utilizing bls (Boneh-Lynn-Shacham) instead of secp256k1 (the original HotStuff implementation), and Kauri, which enhances HotStuff-bls through the incorporation of tree-based message dissemination (and aggregation) along with an improved pipelining scheme.

Results. Figure 7.7 illustrates our findings. In general, we observe that the system throughput results obtained for both implementations are nearly identical. For instance, at $n = 400$, Kauri demonstrates a throughput of 4,518 op/s (compared to the real measurement of 4,584 op/s), while HotStuff-bls achieves 230 op/s (versus the real measurement of 252.67 op/s).

Furthermore, we conducted an evaluation of the consensus latency in Kauri, mirroring the approach in Figure 8 from (Neiheiser, Matos, et al. 2021). We compared this against the scenario with $n = 100$ and a 25 Mbit/s latency. While the real experiment detailed in the Kauri paper reports a latency of 563 ms, in the simulation deciding a block takes at least 585 ms.

7.3.6 Resource Consumption

Furthermore, we conduct an investigation of how resource utilization, specifically memory usage and simulation time, scales with the growing scale of the system. To facilitate this investigation, we

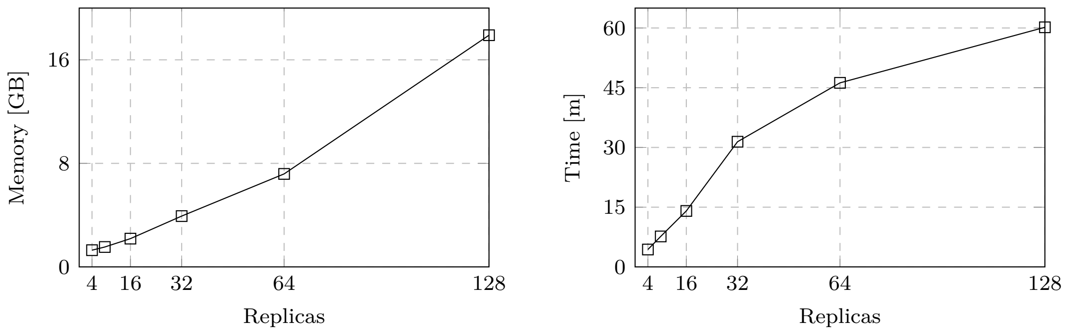


Figure 7.8: Resource consumption on the host machine when running simulations with different system sizes (number of replicas).

employ the HotStuff-secp256k1 “10 ms” simulations, which demonstrate relatively consistent system performance as the system scale increases. These simulations are conducted on an Ubuntu 20.04 virtual machine equipped with 214 GB of memory and 20 threads (of which 16 are allocated for the simulation). The host machine uses an Intel Xeon Gold 6210U CPU running at 2.5 GHz.

Our observations reveal that both active host memory and elapsed time increase proportionally with the increasing system scale (as depicted in Figure 7.8). Notably, the graph illustrating memory utilization in Figure 7.8 exhibits a nearly linear growth pattern. Based on this linear increase in memory consumption, we estimate that a configuration with 512 replicas will necessitate approximately 64 GiB of memory. Consequently, it should be entirely feasible to simulate up to 512 HotStuff replicas on a well-equipped host.

Experiences with Different Programming Languages

Throughout our validity analysis, we tested through various open-source BFT frameworks, as listed in Table 7.1. These frameworks are implemented in different programming languages, including C++, Rust, and Java. Our findings revealed that the Java-based BFT-SMaRt library exhibited the highest memory requirements. However, it’s noteworthy that we were able to successfully simulate a scenario with $n = 128$ replicas on our standard hardware server without encountering any issues. This outcome further reinforces our confidence in the scalability and resource-efficient nature of our methodology.

7.4 Evaluation

In the following section, we conduct a comparative analysis of diverse BFT protocol implementations, considering a range of system sizes and different *scenarios* (as explained next). Our goal is to gain insights into the performance and behavior of these implementations across various scenarios.

Scenarios and Objectives

We begin by benchmarking protocols under ideal, failure-free execution conditions. This evaluation involves increasing the system size while also varying the block size. Ideally, this analysis allows us to compare the performance of the different BFT implementations in an assumed “good-case” deployment. Next, we shift our attention to scenarios that reflect less ideal situations, in particular, we cover the following situations:

- We simulate network scenarios introducing packet loss, replicating a lossy network behavior.
- Furthermore, we investigate the system’s resilience by conducting experiments simulating a *denial-of-service (DOS) attack*. In this scenario, a specific client attempts to overload the system with an excessive number of operations.

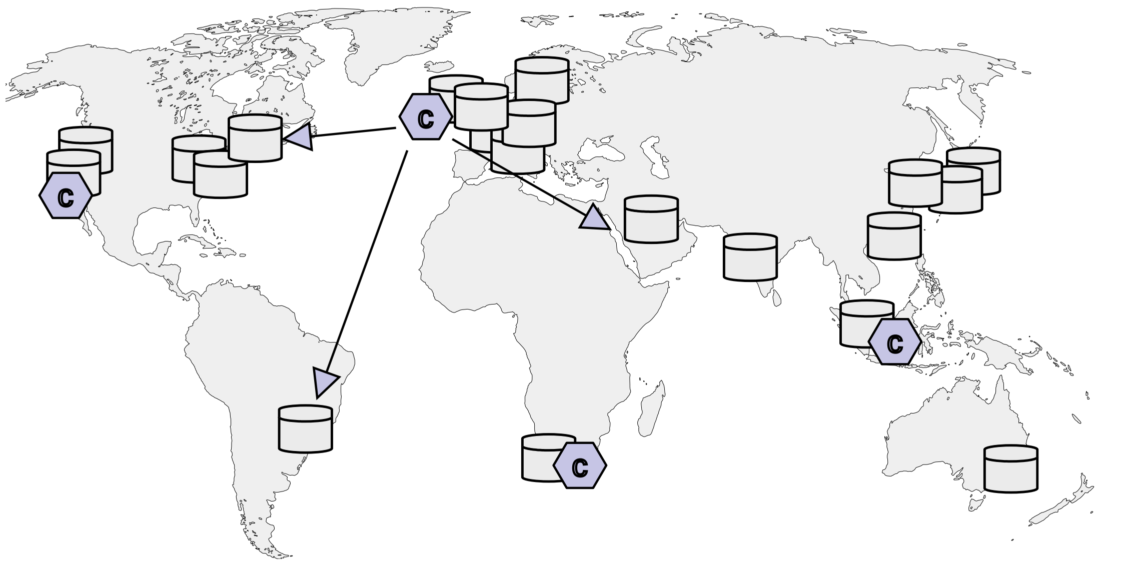


Figure 7.9: The `aws21` map mimics a planetary-scale deployment on the AWS infrastructure, with replicas spread across 21 regions, and clients (c) submitting requests to replicas.

- Lastly, we explore the impact of crash faults in a scenario with induced crash faults (in the leader replica) occurring at a specified point in simulated time. This allows to validate the performance during the recovery mechanism.

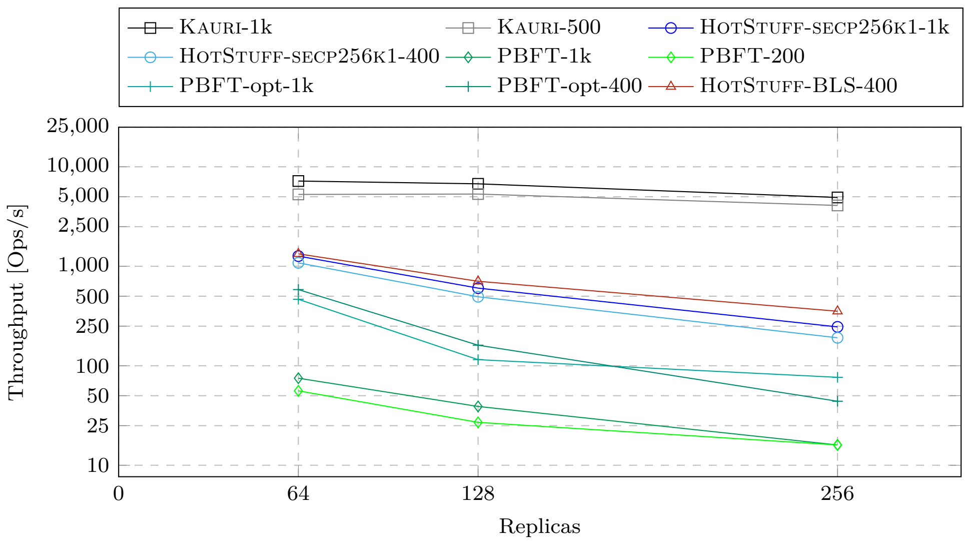
7.4.1 Experimental Setup

Environment. Our controlled experimental environment is built upon the simulation of a heterogeneous planetary-scale network, comprising 21 regions sourced from the AWS cloud infrastructure, as illustrated in Figure 7.9. The latencies in this environment are derived from real-world latency statistics using the *cloudping* component integrated into our frontend.

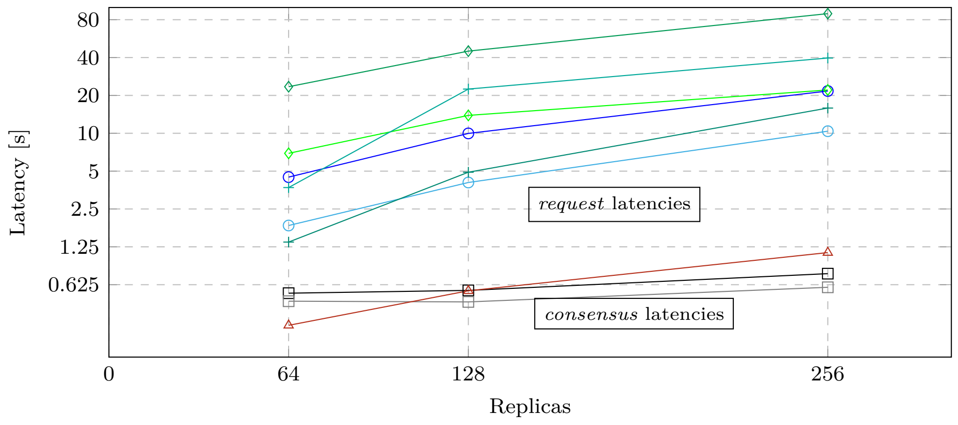
Distribution of replicas and clients. In our experiments, we may use varying numbers of replicas. In each experiment, these replicas are evenly distributed across all regions within the simulated network. Concerning bandwidth, we set a rate of 25 Mbit/s, aligning with the practice in many related research studies that aim to model commodity hardware within planetary-spanning networks (e.g., as seen in the `global` configuration of (Neiheiser, Matos, et al. 2021)). The geographic location of clients is also indicated by Figure 7.9.

Workload. Furthermore, in our experiments, we have the flexibility to employ different numbers of clients to submit requests to the replicas. Specifically, we choose the number of clients and concurrent requests in a manner that optimally saturates the observable system throughput. This saturation point is determined by ensuring that the number of concurrently submitted requests is sufficient to meet two key criteria: (1) filling the block size of each block processed by a BFT protocol in parallel, and (2) maintaining a block size of pending requests at the leader, allowing the immediate dissemination of the next block as soon as an ongoing consensus is finalized.

Defaults in protocol configurations. In our simulations, we employ the following protocols: PBFT, HotStuff-secp256k1, HotStuff-bls, and Kauri. The operations being processed carry a payload of 500 bytes, which is approximately equivalent to the average size of a Bitcoin transaction. By default, the block size is set to 1000 operations, unless explicitly specified otherwise (we may experiment with varying block sizes to tune the performance of individual protocols).



(a) Throughput.



(b) Latency.

Figure 7.10: Performance of BFT protocol implementations in a geo-distributed and fault-free scenario using 25 Mbit/s network links.

Recorded protocol run time. Each simulation deploys replicas and clients and then runs the BFT protocol for at least 120 seconds of simulated time within the environment.

7.4.2 Failure-Free Scenario: Scalability and Block Size

In our initial experiment, we evaluate the performance of the BFT protocols within our simulated environment, assuming a fault-free scenario. Moreover, we conduct multiple iterations of each simulation, systematically scaling up the system size (specifically, using a total of 64, 128, and 256 replicas) and adjusting the block size. We consider both a default block size of 1000 operations and an optimized block size tailored to each individual protocol to optimize for lower latency. These variations in block size for a protocol configuration are indicated by appending the "-{blockSize}" postfix to the respective protocol name.

Results

Our findings are illustrated in Figure 7.10 (which uses log y-axes in the diagrams). Notably, the simulation results indicate remarkable differences in the performance of BFT protocols, with performance metrics spanning several orders of magnitude. Among all the protocol implementations,

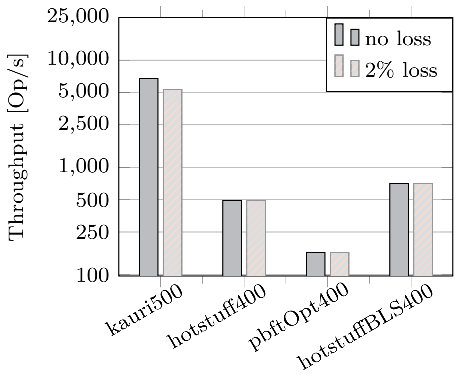


Figure 7.11: Effect of packet loss.

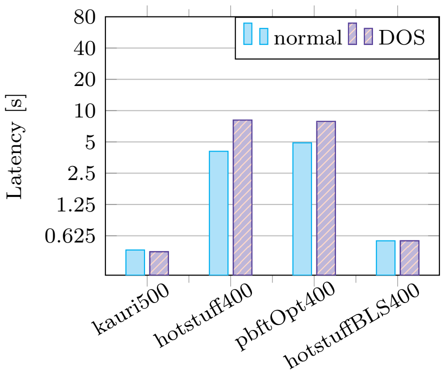


Figure 7.12: Effect of a DoS attack.

KAURI-1k stands out as the best performer. For instance, at $n = 64$, it achieves a throughput of 7192 op/s with a latency of 535 ms. When the system size reaches $n = 256$, Kauri demonstrates an impressive nearly 20-fold increase in throughput compared to HOTSTUFF-SECP256K1 within our heterogeneous environment, achieving 4917 op/s compared to 246 op/s. It is worth noting that evaluations of Kauri have reported a potential increase of up to 28 times over HotStuff in scenarios where communication link latencies are homogeneously distributed (Neiheiser, Matos, et al. 2021).

We also note a notably lower performance exhibited by the evaluated PBFT-1k implementation. Specifically, at $n = 64$, it achieves a throughput of 75 op/s with a latency of 23.45 s. The critical issue impairing performance with this implementation lies in its inlining of full operations within blocks, leading to large blocks and thus prolonged dissemination delays when transmitted over the restricted 25 Mbit/s network links. In contrast, other protocol implementations adopt a more efficient approach by including only SHA-256 hashes of operations within the blocks. This strategy significantly accelerates the dissemination time of block proposals.

Optimized PBFT. To ensure a fair comparison, we mimic the PBFT implementation as if it were using the *big request* optimization (see Section 2.3.2), denoted as PBFT-opt (Castro and Liskov 1999). Under this configuration, it achieves a throughput of 466 op/s and a latency of 3.7 s at $n = 64$. However, it's important to note that HOTSTUFF-SECP256K1 still maintains a performance advantage over the PBFT implementation we employed. This advantage stems from HOTSTUFF-SECP256K1's utilization of pipelining and the reuse of quorum certificates during aggregation, which can enhance performance in this particular scenario.

BLS vs. SECP256k1 signatures. Interestingly, we observe a significant increase in throughput at $n = 256$ for the BLS implementation of HOTSTUFF-SECP256K1-BLS-400, with a boost of 1.85× compared to its counterpart using SECP256K1 (353 op/s versus 191 op/s).

Varying the block size. Adjusting the block size has a noticeable impact on the observed latency. Smaller blocks can be more quickly disseminated which then decreases latency, in particular, if clients operate in a closed loop, i.e., only issue a constant number of k operations concurrently then wait for obtaining responses and completing pending operations before submitting new operations.

7.4.3 Packet Loss

In our subsequent experiment, we investigate how system throughput is influenced by network links prone to packet loss. To conduct this analysis, we maintain the same environment as described in the previous section, with a constant system size of $n = 128$ replicas. However, we introduce a packet loss rate of 2% on every network link.

Results

Our simulation results, presented in Figure 7.11, indicate consistent protocol performance for PBFT and HotStuff-secp256k1, with only a slight decrease in throughput observed for Kauri. Based on these findings, it appears that a small packet loss of 2% has only a limited impact on the performance of BFT protocols. This is probably due to the consistent use of TCP by all of the protocol implementations, which tolerates packetloss to a fair extent.

7.4.4 Denial-of-Service (DoS) Attack

In this experiment, our objective is to assess the resilience of BFT protocol implementations to deliberate attempts by specific clients to overwhelm the system. We maintain the standard setup using $n = 128$ replicas for this evaluation. To simulate system overload, we introduce a designated "malicious" client that is responsible for attempting a denial-of-service attack and submits a significantly larger number of outstanding operations to the system. Specifically, this client submits 10× more operations than in Section 7.4.2 within a brief time frame of 120 seconds. We aim to assess how this increased load affects the request latency experienced by correct clients.

Results

Our simulation results, presented in Figure 7.12, reveal that an increase in the number of outstanding client requests leads to higher observed latency in both PBFT (from 4.9 s to 15.2 s) and HotStuff-secp256k1 (from 4.1 s to 8.2 s). This latency increase is due to the queuing of submitted requests, which must wait for a growing amount of time to be processed.

Interestingly, we observed nearly identical latency results for Kauri and HotStuff-bls. Upon closer inspection of their implemented benchmark applications, we realized that these implementations report only the consensus latency of replicas and not the end-to-end request latency experienced by clients[7]. Consequently, the latency results obtained may not be directly comparable for this experiment. It is worth noting that different BFT protocol implementations may use slightly different metrics in their benchmark suites. While this requires caution when comparing results, it does not hinder our general approach.

It is also noteworthy that none of the tested implementations seemed to have mechanisms in place to prevent system overload, such as limiting the number of requests accepted from a single client.

7.4.5 Crashing Replicas

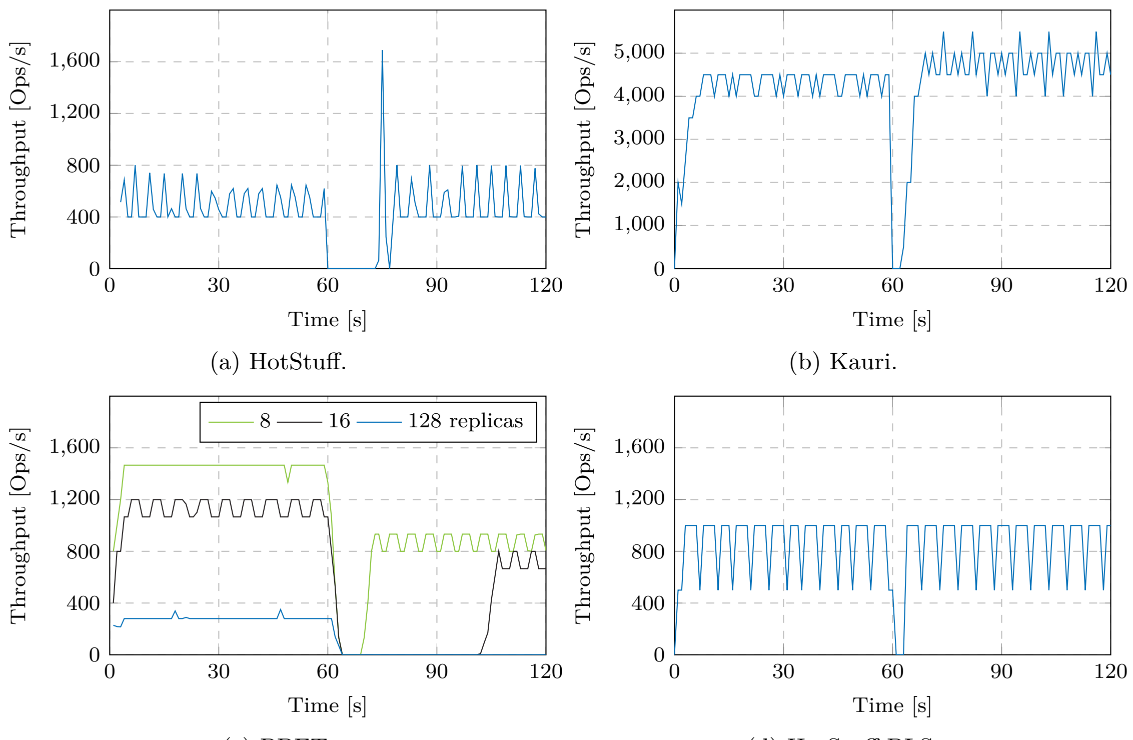
In our final experiment, we assess the crash fault tolerance of the protocol implementations using our standard setup of $n = 128$ replicas. We intentionally induce a crash fault at the leader replica at a specific time, $\tau = 60$ s, to observe how the protocols recover their performance.

During the simulations, we encountered an issue with the PBFT implementation we used (we used the Rust-based implementation from `github.com/ibr-ds/themis` and not the official from Castro et al.). Notably, the *view change* mechanism did not function as expected. We reached out to the developers and received a patch, as a recent commit had been missing from the public GitHub repository. This patch resolved the issue, at least for smaller system sizes. This situation shows how our methodology can be also helpful in detecting protocol implementation bugs.

Results

Our results for the crash fault resilience experiment are presented in Figure 7.13. It's important to note that the failover time of a protocol is generally influenced by its timeout parameterization,

[7]This means these latency results do not include client to replica communication or request queuing (wait time) at the leader before being proposed in a block.



(a) HotStuff. (b) Kauri.

(c) PBFT. (d) HotStuff-BLS.

Figure 7.13: Inducing a single crash fault in the leader.

and it's possible that tighter timeouts could have been employed. For instance, it is mentioned in (Neiheiser, Matos, et al. 2021) that Kauri can use more aggressive timeout values[8] compared to HotStuff. Interestingly, we observed some noteworthy behaviors in the protocols.

In HotStuff-secp256k1, after the failover, a new HotStuff leader pushes a larger block which leads to an observable throughput spike (to over 1600 op/s): The leader tries to commit this large block quickly by building a three-chain in which the large block is followed by empty blocks (now, this leads to a short, second throughput drop to 0). Subsequently, the throughput of the HotStuff protocol run stabilizes. Interestingly, HotStuff-bls does not display such phenomena, which suggests a change in behavior of its employed Pacemaker.

In contrast, our observations show that the new leader in Kauri can recover protocol performance more rapidly than in HotStuff. Additionally, the throughput level appears to be slightly higher. However, this may be due to the new leader being located in a more favorable region of the world, as the leader's geographic location can impact BFT protocol performance.

The PBFT implementation successfully completed its failover mechanism (the view change) only for smaller system sizes and experienced a longer failover time compared to HotStuff and Kauri. Upon closer examination of the implementation's behavior, we identified an inefficient approach in the implementation related to the utilization of bandwidth. In this particular PBFT implementation, when a request timed out, the new leader would assign a sequence number to it and promptly propose it in a new block, which contained only a single request. This approach results in increased costs for running the agreement protocol within a WAN. This way, the costs of running the agreement protocol in the WAN (with notably low bandwidth) would not amortize over multiple (hundreds of) operations because every timed-out operation required collecting its individual quorum certificate.

[8]The impeach timer in HotStuff was 10 s per default for the `/hot-stuff/libhotstuff` GitHub implementation and 3 s per default in the implementation of `/Raycoms/Kauri-Public` (Kauri and HotStuff-bls).

	(Singh et al. 2008)	(P.-L. Wang et al. 2022)	This work
plug & play	✗	✗	✓
scalability	✗	✓	✓
resource friendly	✓	✓	✓
asses throughput	✓	✗	✓
assess latency	✓	✓	✓

Table 7.2: Simulation methods for BFT protocol research.

7.5 Additional Related Work

In this section, we give an overview of research related to the field of simulating BFT protocols.

Simulation or Modeling of BFT Protocols

There are already existing methods developed exclusively for simulating BFT protocols. A brief comparison between the method presented in this thesis and its two main competitive methods, BFTSim (Singh et al. 2008) and the tool of (P.-L. Wang et al. 2022) is presented in Table 7.2.

BFTSim. The first simulator explicitly designed for traditional BFT protocols, such as PBFT (Castro and Liskov 1999) or Zyzzyva (Kotla et al. 2007), is BFTSim (Singh et al. 2008). However, BFTSim has certain limitations that make it less practical for newer, "blockchain envisioned" BFT protocols. It is primarily tailored for small replica groups, and its limited scalability poses challenges when simulating larger systems. Furthermore, BFTSim requires modeling BFT protocols using the P2 language, which can introduce errors due to the complexity of protocols like PBFT, with its complex view change mechanism, and Zyzzyva, which involves numerous corner cases. While BFTSim is capable of simulating faults, it focuses solely on non-malicious behavior like crash faults, lacking the functionality to address more sophisticated attacks. One of BFTSim's strengths is its use of ns-2 for realistic networking simulation, and it is resource-friendly, and capable of running on a single machine.

In contrast to BFTSim, our method allows us to experiment with *real protocol implementations*. We believe this has two main benefits: First, in some sense, it strengthens the validity of simulation results, because no error-prone re-implementation or modeling of a (very complex) BFT protocol is needed. Instead, the source code of the implementation serves as the single reference point (i.e., following the principle of *code is law*). This advantage also means that we *can save a lot of time* and evaluate a larger number of BFT protocols without much effort. It is easy for the developer of a new BFT protocol to adapt our method because it only requires writing a new *protocol connector* and subsequently, the developer can compare his implementation against several already supported BFT protocols in pre-defined benchmarking scenarios. Second, the presented methodology improves on the scalability aspect because it can run with hundreds of replicas which seems to be impossible for BFTSim (according to the experimentation of (P.-L. Wang et al. 2022) only up to 32 PBFT replicas and a limited number of clients is supported in BFTSim).

Scalable BFT simulation. Recently, Wang et al. introduced a BFT simulator known for its resource-friendly nature and high scalability (P.-L. Wang et al. 2022). Notably, this simulator includes an "attacker module" that offers predefined attacks, including partitioning, adaptive, and rushing attacks. However, like BFTSim, it also necessitates the re-implementation of a BFT protocol (in this case in JavaScript). One limitation to consider is that this simulator cannot yet measure throughput. Additionally, instead of emulating real network protocols, it uses a high-level model to capture network characteristics. In this model, message delays are represented by a variable sampled from Gaussian or Poisson distributions, which must be defined in advance. Latencies are not derived from a real-world statistical latency map (thus not reflecting heterogeneous latencies as they occur in geographically dispersed deployments).

In contrast to the simulator of Wang et al., the simulation method presented in this thesis does not require a re-implementation of the protocol in JavaScript and can also measure throughput, which is an important performance metric for blockchains. It should be also noted that the goals of the tool of Wang et al. and our tool were somewhat different: The authors of (P.-L. Wang et al. 2022) were interested in analyzing the potential impact of problematic network conditions (such as misconfigured timeouts) and attacks on BFT protocol latency, which makes it suitable for a validation purpose. Our method focuses on mimicking *realistic deployment scenarios* (such as planetary-scale deployments) with authentic latency maps to reason about the system performance (both latency *and throughput*) in real-world blockchain deployments.

Further approaches. Furthermore, related work in behavior prediction includes stochastic modeling of BFT protocols (Nischwitz et al. 2021) and a prior high-level simulation model examined the impact of various message exchange patterns in BFT protocols, yet it falls short in terms of its suitability for analyzing real-world system metrics (F. Silva et al. 2020). Moreover, validation capabilities of BFT protocols have been improved through the generation of unit tests using the Twins methodology (Bano, Sonnino, Chursin, et al. 2022).

To the best of our knowledge, the methodology for evaluating BFT protocols as presented in this thesis is the only one featuring a *plug-and-play* capability, which not only alleviates the evaluation of a broad variety of BFT protocols but also helps strengthen the validity of results, because no error-prone re-implementation is involved.

Generic Simulators and Emulators

Additionally, there are tools available for emulating or simulating a wide range of generic distributed systems. These tools can substantiate powerful building blocks for evaluating BFT protocols.

Emulators. For instance, emulators like Mininet (Lantz et al. 2010; Handigol et al. 2012) and Kollaps (Gouveia et al. 2020) create realistic networks where actual Internet protocols and application code run, with time synchronized to the wall clock. While these approaches offer a high level of realism, they tend to be less resource-friendly. Mininet, initially lacked scalability but addressed this issue with the introduction of Maxinet (Wette et al. 2014), which enables distributed emulation using multiple physical machines. On the other hand, Kollaps (Gouveia et al. 2020) is a scalable emulator but demands a significant number of physical machines for conducting large-scale experiments.

Simulators. Furthermore, ns-3 (Riley and Henderson 2010) is a resource-friendly and scalable network simulator. However, it requires the development of an application model, which can impede application layer realism and hinder plug-and-play utility. In contrast, Phantom (Jansen et al. 2022) adopts a hybrid emulation/simulation architecture. It executes real applications as native OS processes and integrates them into a high-performance network and kernel simulation. This approach allows Phantom to scale to large system sizes while preserving application layer realism.

The method presented in this thesis relies on Phantom (Jansen et al. 2022) to conduct the simulations, because its hybrid design allows for high-performance network simulations. We addressed open challenges that remain when using Phantom for BFT protocols and large-scale systems, such as the creation of realistic network topologies, the bootstrapping of BFT protocols, their runtime artifacts and configuration files, as well as fault induction capabilities aimed at measuring BFT protocol failover behavior, the ability for bulk experimentation to systematically explore BFT protocol parameter space and also resource monitoring.

Blockchain Research

Several simulators have been developed for blockchain research, including Shadow-Bitcoin (Miller and Jansen 2015), the Bitcoin blockchain simulator (Gervais et al. 2016), BlockSim (Faria and Correia 2019), SimBlock (Aoki et al. 2019), and ChainSim (B. Wang et al. 2020).

However, these simulators primarily focus on constructing models that accurately represent the features of Proof-of-Work (PoW) consensus mechanisms. As a result, as of the time of writing, they are less suitable for adoption when conducting BFT protocol research.

The methodology presented in this thesis is explicitly designed for BFT protocol research and is efficient in conducting large-scale simulations.

Tools for Vulnerability Detection

Twins (Bano, Sonnino, Chursin, et al. 2022) is an unit test case generator designed to simulate Byzantine behavior through the replication of cryptographic IDs of replicas, thus producing protocol runs that feature equivocations or forgotten replica states. We plan to explore the integration of Twins within our simulation methodology in future work.

Related work also investigates on checking the liveness of streamlined BFT protocols (Decouchant, Ozkan, et al. 2023): The authors introduced the concepts of *partial state* and *hot state* to represent relevant information about protocol executions. It then employs temperature checking and lasso detection to identify recurring states, thus identifying possible liveness issues in a automated way.

ByzFuzz (Winter et al. 2023) is a fuzzing technique designed to automatically detect bugs in implementations of BFT SMR protocols. For instance, the creators of ByzFuzz demonstrated that with their fuzzing approach, they are able to systematically reproduce the vulnerability discussed in Chapter 4.2.1.

The methodology presented in this thesis aims to measure performance of BFT protocols at large scale. While we are curious about the behavior and performance of systems under uncivil conditions, at the present time, only a few simple scenarios are supported (i.e., replica crashes, package loss on network links or DOS attacks conducted by malicious clients). We see the support for more complex (Byzantine) behavior and attacking scenarios as an interesting direction for future work.

7.6 Concluding Remarks

This thesis introduced a methodology for evaluating the performance of BFT protocols through network simulations. A significant advantage of the proposed approach, in comparison to related methods such as (Singh et al. 2008; P.-L. Wang et al. 2022), is that this methodology is the first one which *seamlessly integrates existing protocol implementations* without the need for error-prone and time-consuming re-implementations in modeling languages. Compared to these two related simulation approaches of BFT protocols we know of, our presented method additionally scales to large system sizes (which the approach of (Singh et al. 2008) does not support) and can additionally evaluate system throughput (evaluating this metric is not possible with the tool of (P.-L. Wang et al. 2022)).

Furthermore, the developed methodology has proven effective in assessing protocol performance as system scale increases and within realistic environments. An envisioned use of this tool is to spot bugs on the implementation level. For instance, we accidentally observed problems in the view change subprotocol of the PBFT implementation that we used and which only became apparent at larger system sizes (see Figure 7.13c). We believe simulation-based methods can generally assist in validating BFT protocol implementations and want to explore their capabilities more in future work.

In summary, our simulation-based evaluation method offers a handy tool for researchers and practitioners working with BFT protocols in the context of DLT applications. The tool not only simplifies scalability analysis but also provides cost-effectiveness and accuracy, thus facilitating the design and deployment of more efficient and resilient BFT-based distributed systems. For instance, we envision that the tool could be integrated into a CI/CD pipeline and conduct integration tests (at large scale) of real system software every time a new commit happens, thus validating whether recent changes impacted the performance of a BFT protocol implementation.

Future Work

We plan to extend the scope of the simulation methodology in several ways in future work. First, we want to extend our evaluations towards multi-leader and leaderless BFT protocols as well as towards asynchronous BFT protocols. Our goal is to study the scalability aspect not only under the commonly used “failure-free” evaluation setups but also in “uncivil” conditions. Second, we want to expand the fault induction abilities of our method to allow for more complex behavior, for instance, malicious attacks carried out by a fraction of Byzantine replicas such as equivocations. A further interesting aspect might be the impact of longer communication link breaks (such that TCP connections fail), to see how well such situations are handled in protocol implementations.

Chapter 8
Conclusions

This thesis tackled questions that seek to improve the suitability of BFT SMR for planetary-scale systems by reducing its latency through optimizations at the protocol level and enabling the protocol to respond to environmental changes. For this purpose, the thesis targeted three concrete topics involving the optimization of a BFT SMR system:

1. the correct integration of fast, live, and linearizable read-only operations into BFT SMR,
2. the design of an adaptive weight-enabled BFT SMR system that optimizes consensus latency during runtime and can respond to environmental changes,
3. and the tentative use of smaller consensus quorums in an adaptive BFT SMR system with self-auditing capabilities;

Apart from these topics that focus on optimizations, the thesis developed a methodology that alleviates the evaluation of BFT SMR protocols in planetary-scale deployments through the co-opting of Linux processes into a simulated network environment.

For the correct integration of read-only operations, this thesis found a crucial liveness bug in optimized PBFT (and other protocols that followed its blueprint) that allowed a malicious leader to censor correct clients. The thesis proposed two possibilities to patch the vulnerability and analyzed linearizability and liveness properties under the proposed protocol modifications. This theoretical analysis was followed by an experimental evaluation of the performance of an optimized BFT SMR system.

For the design of an adaptive and weight-enabled BFT SMR system, the thesis introduced AWARE, a protocol that adds a degree of environmental awareness to the replicated system through its inbuilt self-monitoring mechanism and continuously strives for latency improvements by assigning high weights to the best-connected replicas. The design of AWARE was studied in a series of experiments using planetary-scale deployments on the AWS cloud infrastructure, demonstrating a strong potential for achieving latency improvements in micro benchmarks and when integrated as an ordering service in the Hyperledger Fabric blockchain platform.

For the tentative use of smaller consensus quorums in an adaptive BFT SMR system with self-auditing capabilities, we showed how these smaller quorums can be utilized in a safe way through a novel transformation called FlashConsensus. In particular, FlashConsensus integrates BFT protocol forensics and two modes of operation into a quorum-based BFT SMR protocol just like AWARE to benefit from the speedup achievable with weighted replication when using a smaller resilience threshold for building the underlying quorum system. Notably, FlashConsensus does not compromise on the optimal ($t = \lfloor \frac{n-1}{3} \rfloor$)-resilience.

As a side-product, this thesis introduced a novel method that simplifies and accelerates the evaluation of BFT protocol implementations using high-performance network simulations. In comparison with the alternative simulation-based approaches, this method does not require any modifications or re-implementation efforts which are error-prone and time-consuming. The method may further serve as a handy validation tool to detect bugs during the development phase of BFT protocols.

8.1 Contributions

Fast and correct reads in BFT SMR. PBFT pioneered the field of BFT SMR protocols delivering performance akin to that of non-replicated systems in practical scenarios. This impressive performance can be attributed to several optimizations introduced by PBFT, one of which involves *read-only* requests. These specific client requests sidestep the three-step agreement protocol, enabling replicas to respond directly. In a surprising revelation, we unveil a vulnerability in the well-studied PBFT, specifically related to the read-only request optimization that was introduced over two decades ago. This vulnerability, strikingly, has the potential to compromise the liveness of the protocol, affecting also standard, totally-ordered operations. To show this vulnerability, we present an attack strategy called *isolating-leader* attack, in which a malicious leader censors correct clients. To patch the vulnerability we developed two solutions that ensure both liveness and linearizability of read-only-optimized BFT SMR. We implemented the solutions in the BFT-SMaRt framework and conducted an experimental evaluation of the protocol behavior when under attack.

Adaptive Wide-Area Replication. In geographically distributed BFT SMR systems, the speed of consensus is fundamentally limited by the quorum formation rules and heterogeneous communication links of replicas. While the WHEAT protocol demonstrated that planetary-scale deployments can benefit from *weighted replication* to increase flexibility in quorum formation, employing weights necessitates a solution to find the ideal weight distribution *during runtime* thus adjusting the system to its environmental conditions. AWARE enhances the geographic scalability of consensus in planetary-scale systems by autonomously selecting a weight distribution and leader position that optimizes system performance based on the network capabilities of replicas. This optimization aids in the emergence of faster quorums within the system. AWARE integrates a reliable self-monitoring procedure to gather information on the speed of network links. Subsequently, a consensus latency prediction model determines the optimal system configuration for the next scheduled reconfiguration. Moreover, we implemented AWARE on top of WHEAT and conducted experiments spanning multiple AWS EC2 regions to validate that AWARE can dynamically optimize consensus latency. Our experiments include experimentation with the blockchain platform Hyperledger Fabric using AWARE as a consensus substrate to order transactions, i.e., achieving agreement on which block is appended next to the ledger. In this experiment, frontends observed a speedup of up to 1.43×.

FlashConsensus. A persistent challenge in the development of AWARE has revolved around striking the right balance between resilience and performance. A significant observation we made was that by lowering the resilience threshold t, smaller, *compacter* consensus quorums emerge that lead to faster consensus decisions in planetary-scale systems. Yet, since the choice of t had to be made before deploying the system and could not be altered afterward, it introduced an inherent trade-off between resilience and performance. With FlashConsensus, we adopt a novel approach: Deliberately underestimate the resilience threshold, using a lower threshold $t_{fast} < t$ to leverage this advantage of smaller, faster quorums. To make this approach work in practice, we found that the BFT SMR protocol must be augmented with detection and self-repair mechanisms. FlashConsensus can transition between an optimized execution and the execution of a resilient protocol when optimal resilience ($t = \lfloor \frac{n-1}{3} \rfloor$) is essential to maintain system correctness. We also gained novel insights when experimenting with a form of client-side speculation that can lower client-observed latencies even further by letting clients access incremental consistency levels. In an experimental evaluation, which involved up to 51 replicas scattered across the globe, FlashConsensus demonstrated its capability to run finalized operations in under 0.4 seconds (and reaching a consensus latency of a mere 88 ms). From the perspective of clients, the speed of FlashConsensus significantly outpaces a PBFT-like protocol running in the same network environment by between almost 2× (for linearizable operations) up to 6× (when using the highest degree of speculation).

Simulation of BFT SMR implementations. This thesis introduced an evaluation methodology for assessing the performance of BFT SMR protocols using network simulations. A significant advantage of our approach is its seamless integration of existing protocol implementations. This eliminates the need for error-prone re-implementations in modeling languages. Our methodology has demonstrated its effectiveness in evaluating protocol performance as the system scale (both the geographic dimension and number of replicas) increases, all within realistic environments. Furthermore, it serves as a valuable tool for pinpointing implementation bugs since simulations can conduct extensive integration tests of distributed systems cost-effectively. To summarize, our simulation-based evaluation method provides a practical tool for both researchers and practitioners working with BFT protocols in the context of DLT applications. It streamlines scalability analysis, offers cost-effectiveness, and ensures a reasonable degree of accuracy, thereby simplifying the design and deployment of more efficient and resilient BFT-based distributed systems in the future.

8.2 Future Work

The approaches presented in this thesis targeted to lower the latency of systems. This was an especially interesting topic since latency improvements can be primarily achieved through optimizations at the protocol level because the speed of communication links tends to approach limits that are dictated by the laws of physics. In particular, the Internet speed can not become arbitrarily fast. Yet, we see also potential for *adaptiveness* in many BFT protocols that are more focused on improving *throughput* instead of latency such as Kauri (Neiheiser, Matos, et al. 2021). Kauri uses a balanced tree for message dissemination, and, given a specific planetary-scale deployment of replicas, one could think on how to craft optimized trees to improve performance for such a deployment scenario.

To improve the scalability of the prediction model in AWARE, the generic heuristic, *simulated annealing* is used. Mastering the configuration space of BFT SMR protocol in the presence of planetary-scale environments becomes quite complex, and may be a suitable task for a *machine learning* (ML) algorithm. In the future, we would like to explore how ML can assist in optimizing the performance of BFT SMR deployments further by letting it parametrize several protocol-specific options.

Furthermore, we intend to broaden the scope of our simulation methodology in several ways. Firstly, we aim to extend our evaluations to encompass multi-leader and leaderless BFT protocols, as well as novel asynchronous BFT protocols. Moreover, we want to explore scalability not only in typical scenarios but also in more challenging and adversarial conditions. For this purpose, our method's fault induction capabilities need to include more complex behaviors, such as malicious attacks carried out by a subset of Byzantine replicas, including equivocations. Additionally, we are interested in exploring the impact of longer communication link disruptions, such as TCP connection failures, to assess how well BFT protocol implementations can handle such scenarios.

Publication List

This chapter provides an overview of my collaborative publications, and presents concise descriptions and a references for each paper that is related to the thesis. Furthermore, it details the role and contribution of each individual co-author, as defined per the Contributor Roles Taxonomy (CRediT1[1]), which can be found attached at the end of this chapter.

Paper 1

This conference paper proposes a novel mechanism called Adaptive Wide Area Replication (AWARE), a technique to automatically optimize a weighted-enabled BFT SMR system for operation in a wide-area network, thus achieving low latency. This paper relates to Chapter 5 of the thesis.

Christian Berger, Hans P. Reiser, João Sousa, and Alysson Bessani (2019). "Resilient Wide-Area Byzantine Consensus Using Adaptive Weighted Replication". In: *Proceedings of the 38th IEEE Symposium on Reliable Distributed Systems (SRDS)*

Author	**Roles**
Christian Berger	conceptualization, methodology, software, investigation, validation, data curation, visualization, writing
Hans P. Reiser	conceptualization, supervision, writing - review and editing
João Sousa	writing - review and editing
Alysson Bessani	conceptualization, supervision, writing

Paper 2

This journal paper is the extended version of Paper 1. It augments the proposed AWARE by additional experimentation with the blockchain system Hyperledger Fabric and additionally proposes a heuristic to efficiently traverse the search space of system configurations necessary to improve the scalabiltiy of the optimization method for larger replia groups. This paper relates to Chapter 5 of the thesis, too.

Christian Berger, Hans P. Reiser, João Sousa, and Alysson Bessani (2022). "AWARE: Adaptive Wide-Area Replication for Fast and Resilient Byzantine Consensus". In: *IEEE Transactions on Dependable and Secure Computing* 19.3, pp. 1605–1620. DOI: `10.1109/TDSC.2020.3030605`

[1]`https://onlinelibrary.wiley.com/doi/epdf/10.1002/leap.1210`

Author	**Roles**
Christian Berger	conceptualization, methodology, software, investigation, validation, data curation, visualization, writing
Hans P. Reiser	conceptualization, supervision, writing - review and editing
João Sousa	writing - review and editing
Alysson Bessani	conceptualization, supervision, writing

Paper 3

This conference paper identifies a long existing vulnerability in seminal BFT works such as PBFT and BFT-SMaRt that emerges from the integration of the read-only optimization. It also shows how to integrate fast and correct read operations within PBFT (or a similiar BFT system) without compromising liveness. This paper relates to Chapter 4 of the thesis.

> Christian Berger, Hans P. Reiser, and Alysson Bessani (2021). “Making Reads in BFT State Machine Replication Fast, Linearizable, and Live”. In: *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pp. 1–12. DOI: `10.1109/SRDS53918.2021.00010`

Author	**Roles**
Christian Berger	conceptualization, methodology, software, investigation, validation, data curation, visualization, writing
Hans P. Reiser	supervision, writing - review and editing
Alysson Bessani	supervision, writing - review and editing

Paper 4

This workshop paper proposes a methodology to accelerate the evaluation of novel BFT protocol implementations through network simulations. The paper explores the validity of the results obtained from simulations with real-world experimentation and the scalability of the simulation method. This paper is also part of Sadok Ben Toumias Bachelor thesis, which I supervised. Sadok mainly did the implementation (software) part, and a first round of experimentation. This paper relates to Chapter 7 of the thesis.

> Christian Berger, Sadok Ben Toumia, and Hans P. Reiser (2022). “Does My BFT Protocol Implementation Scale?” In: The 3rd International Workshop on Distributed Infrastructure for Common Good (DICG). Quebec, Quebec City, Canada: Association for Computing Machinery, pp. 19–24. ISBN: 9781450399289. DOI: `10.1145/3565383.3566109`. URL: `https://doi.org/10.1145/3565383.3566109`

Author	Roles
Christian Berger	conceptualization, methodology, software, investigation, validation, data curation, visualization, writing
Sadok Ben Toumia	methodology, software, validation, investigation
Hans P. Reiser	supervision, writing - review and editing

Paper 5

Christian Berger, Sadok Ben Toumia, and Hans P. Reiser (2023). “Scalable Performance Evaluation of Byzantine Fault-Tolerant Systems Using Network Simulation”. In: *The 28th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC).* IEEE

Author	Roles
Christian Berger	conceptualization, methodology, software, investigation, validation, data curation, visualization, writing
Sadok Ben Toumia	methodology, software, validation, investigation
Hans P. Reiser	supervision, writing - review and editing

This conference paper extends paper 4, by introducing additional capabilities to the simulation methodology (such as exploring the protocol behavior under induced failures), and, conducting a eye-to-eye comparison of multiple BFT protocols under a realistic “blockchain”-scenario in a simulated WAN environment. This paper relates to Chapter 7 of the thesis.

Paper 6

Christian Berger, Lívio Rodrigues, Hans P. Reiser, Vinicius Cogo, and Alysson Bessani (2024). “Chasing Lightspeed Consensus: Fast Wide-Area Byzantine Replication with Mercury”. In: *The 25th ACM/IFIP International Middleware Conference.* ACM

Author	Roles
Christian Berger	conceptualization, methodology, software, investigation, validation, data curation, visualization, writing
Lívio Rodrigues	conceptualization, methodology, software, investigation, validation, writing - review and editing
Hans P. Reiser	supervision, writing - review and editing
Vinícius Cogo	supervision, writing - review and editing
Alysson Bessani	conceptualization, writing, supervision

This paper is currently under submission at a conference. It proposes FlashConsensus, a novel transformation for weighted BFT SMR systems to further improve latency by tentatively utilizing compacter consensus quorums in a so called “fast mode”, and implementing incremental consistency levels for client-side speculation. FLASHCONSENSUS maintains linearizability under optimal resilience by incorporating two modes of execution and a self-auditing procedure based on BFT forensic techniques. This paper is also part of Lívios Master thesis, who I supervised. Lívio mainly worked on the correct integration of the BFT auditing features into the AWARE prototype and he conducted a first round of experiments in a local testbed. Nonetheless, I also explained the auditing procedure in my thesis to ensure the overall approach is understandable. This paper relates to Chapter 6 of the thesis.

Contributor Roles Taxonomy (CRediT)

Role	**Description**
Conceptualization	Ideas; formulation or evolution of overarching research goals and aims
Methodology	Development or design of methodology; creation of models
Software	Programming, software development; designing computer programs; implementation of the computer code and supporting algorithms; testing of existing code components
Validation	Verification, whether as a part of the activity of separate, of the overall replication/reproducibility of results/experiments and other research outputs
Formal analysis	Application of statistical, mathematical, computational, or other format techniques to analyze or synthesize study data
Investigation	Conducting a research and investigation process, specifically performing the experiments, or data/evidence collection
Resources	Provision of study materials, reagents, materials, patients, laboratory samples, animals, instrumentation, computing resources, or other analysis tools
Data curation	Management activities to annotate (produce metadata), scrub data and maintain research data (including software code where it is necessary for interpreting the data itself) for initial use or later re-use
Writing – original draft	*Preparation, creation and/or presentation of the published work, specifically writing the initial draft (including substantive translation)*
Writing – review & editing	Preparation, creation and/or presentation of the published work by those from the original research group, specifically critical review, commentary or revision – including pre- and post-publication stages
Visualization	Preparation, creation and/or presentation of the published work, specifically visualization/data presentation
Supervision	Oversight and leadership responsibility for the research activity planning and execution, including mentorship external to the core team

List of Abbreviations

AWARE Adaptive Wide Area Replication

AWS Amazon Web Services

BLS Refers to the Boneh–Lynn–Shacham signature scheme

BFT Byzantine Fault Tolerance

BFT-SMaRt Byzantine Fault-Tolerant State Machine Replication

DLT Distributed Ledger Technology

DOS Denial of Service

EC2 Elastic Compute Cloud

EDCSA Elliptic Curve Digital Signature Algorithm

GST Global Stabilization Time

FIFO First-In-First-Out

HLF Hyperledger Fabric

LAN Local Area Network

PBFT Practical Byzantine Fault Tolerance

POC Proof of Culpability

PPP Perfect Point-to-Point Link

QC Quorum Certificate

RBC Reliable Broadcast

RSM Replicated State Machine

SECP256k1 Refers to a specific elliptic curve

SHA Secure Hash Algorihm

SMR State Machine Replication

TLS Transport Layer Security

VM Virtual Machine

WAN Wide Area Network

WHEAT Weight-Enabled Active Replication

List of Algorithms

List of Figures

List of Tables

Bibliography

Abd-El-Malek et al. 2005

Abd-El-Malek, Michael, Gregory R Ganger, Garth R Goodson, Michael K. Reiter, and Jay J Wylie (2005). “Fault-scalable Byzantine fault-tolerant services”. In: *ACM SIGOPS Operating Systems Review* 39.5, pp. 59–74.

Abraham, Gueta, et al. 2017

Abraham, Ittai, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin (2017). “Revisiting fast practical Byzantine fault tolerance”. In: *arXiv preprint arXiv:1712.01367*.

Abraham, Malkhi, et al. 2019

Abraham, Ittai, Dahlia Malkhi, and Alexander Spiegelman (2019). “Asymptotically optimal validated asynchronous Byzantine agreement”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pp. 337–346.

Alqahtani and Demirbas 2021

Alqahtani, Salem and Murat Demirbas (2021). “BigBFT: A Multileader Byzantine Fault Tolerance Protocol for High Throughput”. In: *IEEE Int. Performance, Computing, and Communications Conf. (IPCCC)*, pp. 1–10. DOI: `10.1109/IPCCC51483.2021.9679432`.

Amir, Coan, et al. 2010

Amir, Yair, Brian Coan, Jonathan Kirsch, and John Lane (2010). “Prime: Byzantine replication under attack”. In: *IEEE Transactions on Dependable and Secure Computing* 8.4, pp. 564–577.

Amir, Danilov, et al. 2010

Amir, Yair, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage (Jan. 2010). “Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks”. In: *IEEE Transactions on Dependable and Secure Computing* 7.1, pp. 80–93. ISSN: 1545-5971. DOI: `10.1109/TDSC.2008.53`.

Androulaki et al. 2018

Androulaki, Elli et al. (2018). “Hyperledger Fabric: a distributed operating system for permissioned blockchains”. In: *Proceedings of the 13th EuroSys Conference*. ACM.

Aoki et al. 2019

Aoki, Yusuke, Kai Otsuki, Takeshi Kaneko, Ryohei Banno, and Kazuyuki Shudo (2019). “Simblock: A blockchain network simulator”. In: *IEEE Conf. on Computer Communications Workshops*. Washington, DC, USA: IEEE Comp. Soc., pp. 325–329.

Arun and Ravindran 2020
Arun, Balaji and Binoy Ravindran (2020). “DuoBFT: Resilience vs. Efficiency Trade-off in Byzantine Fault Tolerance”. In: *preprint arXiv:2010.01387*.

Attiya and Welch 1994
Attiya, Hagit and Jennifer L. Welch (May 1994). “Sequential Consistency versus Linearizability”. In: *ACM Transactions on Computer Systems (TOCS)* 12.2, pp. 91–122. ISSN: 0734-2071. DOI: `10.1145/176575.176576`.

Aublin, Guerraoui, et al. 2015
Aublin, Pierre-Louis, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić (2015). “The next 700 BFT protocols”. In: *ACM Transactions on Computer Systems (TOCS)* 32.4, pp. 1–45.

Aublin, Mokhtar, et al. 2013
Aublin, Pierre-Louis, Sonia Ben Mokhtar, and Vivien Quéma (2013). “RBFT: Redundant Byzantine fault tolerance”. In: *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, pp. 297–306.

Avižienis et al. 2004
Avižienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl Landwehr (Jan. 2004). “Basic Concepts and Taxonomy of Dependable and Secure Computing”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1, pp. 11–33. ISSN: 1545-5971. DOI: `10.1109/TDSC.2004.2`. URL: `https://doi.org/10.1109/TDSC.2004.2`.

Bahsoun et al. 2015
Bahsoun, Jean-Paul, Rachid Guerraoui, and Ali Shoker (2015). “Making BFT protocols really adaptive”. In: *Proc. of the 29th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, pp. 904–913.

Baker et al. 2011
Baker, Jason, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh (2011). “Megastore: Providing scalable, highly available storage for interactive services”. In: *Proceedings of the Conference on Innovative Data System Research (CIDR)*, pp. 223–234.

Bal 1990
Bal, Henri E. (1990). *Programming distributed systems.* Silicon Press.

Bano, Sonnino, Al-Bassam, et al. 2017
Bano, Shehar, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis (2017). “Consensus in the age of blockchains”. In: *arXiv preprint arXiv:1711.03936*.

Bano, Sonnino, Chursin, et al. 2022
Bano, Shehar, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi (2022). “Twins: BFT Systems Made Robust”. In: *25th Int. Conf. on Principles of Distributed Systems.* Vol. 217. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 7:1–7:29.

Ben Toumia et al. 2022
Ben Toumia, Sadok, Christian Berger, and Hans P. Reiser (2022). “An Evaluation of Blockchain Application Requirements and Their Satisfaction in Hyperledger Fabric”. In: *Distributed Applications and Interoperable Systems.* Ed. by David Eyers and Spyros Voulgaris. Cham: Springer International Publishing, pp. 3–20. ISBN: 978-3-031-16092-9.

Ben-Or 1983 Ben-Or, Michael (1983). “Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols”. In: *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*. PODC ’83. Montreal, Quebec, Canada: Association for Computing Machinery, pp. 27–30. ISBN: 0897911105. DOI: 10.1145/800221.806707. URL: https://doi.org/10.1145/800221.806707.

Ben-Or et al. 2006
Ben-Or, Michael, Elan Pavlov, and Vinod Vaikuntanathan (2006). “Byzantine agreement in the full-information model in o (log n) rounds”. In: *Proceedings of the thirty-eighth annual ACM symposium on Theory of Computing*, pp. 179–186.

Berger, Eichhammer, et al. 2021
Berger, Christian, Philipp Eichhammer, Hans P. Reiser, Jörg Domaschka, Franz J. Hauck, and Gerhard Habiger (Sept. 2021). “A Survey on Resilience in the IoT: Taxonomy, Classification, and Discussion of Resilience Mechanisms”. In: *ACM Computing Surveys* 54.7. ISSN: 0360-0300. DOI: 10.1145/3462513. URL: https://doi.org/10.1145/3462513.

Berger and Reiser 2018a
Berger, Christian and Hans P. Reiser (2018a). “Scaling Byzantine Consensus: A Broad Analysis”. In: *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers.*

Berger and Reiser 2018b
Berger, Christian and Hans P. Reiser (2018b). “WebBFT: Byzantine fault tolerance for resilient interactive web applications”. In: *Distributed Applications and Interoperable Systems: 18th IFIP WG 6.1 International Conference, DAIS 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings 18*. Springer, pp. 1–17.

Berger, Reiser, and Bessani 2021
Berger, Christian, Hans P. Reiser, and Alysson Bessani (2021). “Making Reads in BFT State Machine Replication Fast, Linearizable, and Live”. In: *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pp. 1–12. DOI: 10.1109/SRDS53918.2021.00010.

Berger, Reiser, Hauck, et al. 2022
Berger, Christian, Hans P. Reiser, Franz J Hauck, Florian Held, and Jörg Domaschka (Sept. 2022). “Automatic Integration of BFT State-Machine Replication into IoT Systems”. In: *2022 18th European Dependable Computing Conference (EDCC)*. Los Alamitos, CA, USA: IEEE Computer Society, pp. 1–8. DOI: 10.1109/EDCC57035.2022.00013. URL: https://doi.ieeecomputersociety.org/10.1109/EDCC57035.2022.00013.

Berger, Reiser, Sousa, et al. 2019
Berger, Christian, Hans P. Reiser, João Sousa, and Alysson Bessani (2019). “Resilient Wide-Area Byzantine Consensus Using Adaptive Weighted Replication”. In: *Proceedings of the 38th IEEE Symposium on Reliable Distributed Systems (SRDS).*

Berger, Reiser, Sousa, et al. 2022
Berger, Christian, Hans P. Reiser, João Sousa, and Alysson Bessani (2022). “AWARE: Adaptive Wide-Area Replication for Fast and Resilient Byzantine Consensus”. In: *IEEE Transactions on Dependable and Secure Computing* 19.3, pp. 1605–1620. DOI: 10.1109/TDSC.2020.3030605.

Berger, Rodrigues, et al. 2024
Berger, Christian, Lívio Rodrigues, Hans P. Reiser, Vinicius Cogo, and Alysson Bessani (2024). "Chasing Lightspeed Consensus: Fast Wide-Area Byzantine Replication with Mercury". In: *The 25th ACM/IFIP International Middleware Conference*. ACM.

Berger, Schwarz-Rüsch, et al. 2023
Berger, Christian, Signe Schwarz-Rüsch, Arne Vogel, Kai Bleeke, Leander Jehl, Hans P. Reiser, and Rüdiger Kapitza (2023). "SoK: Scalability Techniques for BFT Consensus". In: *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE.

Berger, Toumia, et al. 2022
Berger, Christian, Sadok Ben Toumia, and Hans P. Reiser (2022). "Does My BFT Protocol Implementation Scale?" In: The 3rd International Workshop on Distributed Infrastructure for Common Good (DICG). Quebec, Quebec City, Canada: Association for Computing Machinery, pp. 19–24. ISBN: 9781450399289. DOI: `10.1145/3565383.3566109`. URL: `https://doi.org/10.1145/3565383.3566109`.

Berger, Toumia, et al. 2023
Berger, Christian, Sadok Ben Toumia, and Hans P. Reiser (2023). "Scalable Performance Evaluation of Byzantine Fault-Tolerant Systems Using Network Simulation". In: *The 28th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE.

Bessani and Alchieri 2014
Bessani, Alysson and Eduardo Alchieri (2014). "A guided tour on the theory and practice of state machine replication". In: *Tutorial at the 32nd Brazilian symposium on computer networks and distributed systems*. Citeseer.

Bessani, Alchieri, et al. 2020
Bessani, Alysson, Eduardo Alchieri, João Sousa, André Oliveira, and Fernando Pedone (2020). "From Byzantine Replication to Blockchain: Consensus is only the Beginning". In: *Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE.

Bessani, Sousa, et al. 2014
Bessani, Alysson, João Sousa, and Eduardo EP 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 (DSN)*, pp. 355–362.

Blum et al. 2020
Blum, Erica, Jonathan Katz, and Julian Loss (2020). "Network-agnostic state machine replication". In: *arXiv preprint arXiv:2002.03437*.

Boneh et al. 2004
Boneh, Dan, Ben Lynn, and Hovav Shacham (2004). "Short signatures from the Weil pairing". In: *Journal of cryptology* 17, pp. 297–319.

Bonniot et al. 2020
Bonniot, Loïck, Christoph Neumann, and François Taïani (2020). "Pnyxdb: a lightweight leaderless democratic Byzantine fault tolerant replicated datastore". In: *Proc. of the 39th IEEE Int. Symp. on Reliable Distributed Systems (SRDS)*, pp. 155–164.

Bonomi et al. 2021
Bonomi, Silvia, Jérémie Decouchant, Giovanni Farina, Vincent Rahli, and Sébastien Tixeuil (2021). "Practical Byzantine reliable broadcast on partially connected networks". In: *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, pp. 506–516.

Bracha 1987 Bracha, Gabriel (1987). "Asynchronous Byzantine agreement protocols". In: *Information and Computation* 75.2, pp. 130–143.

Bravo et al. 2022a
Bravo, Manuel, Gregory Chockler, and Alexey Gotsman (2022a). "Liveness and latency of Byzantine state-machine replication". In: *36th International Symposium on Distributed Computing (DISC 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Bravo et al. 2022b
Bravo, Manuel, Gregory Chockler, and Alexey Gotsman (2022b). "Liveness and latency of Byzantine state-machine replication". In: *Proc. of the 36th Int. Symp. on Distributed Computing (DISC)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 12:1–12:19.

Burrows 2006 Burrows, Mike (2006). "The Chubby lock service for loosely-coupled distributed systems". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 335–350.

Cachin 2009 Cachin, Christian (2009). "Yet Another Visit to Paxos". In: *IBM Research, Zurich, Switzerland, Technical Report RZ3754*.

Cachin, Guerraoui, et al. 2011
Cachin, Christian, Rachid Guerraoui, and Luís Rodrigues (2011). *Introduction to Reliable and Secure Distributed Programming*. 2nd ed. Springer Science & Business Media. ISBN: 978-3-642-15259-7.

Cachin and Vukolić 2017
Cachin, Christian and Marko Vukolić (2017). "Blockchain consensus protocols in the wild". In: *arXiv preprint arXiv:1707.01873*.

Canetti et al. 1999
Canetti, Ran, Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin (1999). "Adaptive security for threshold cryptosystems". In: *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*. Springer, pp. 98–116.

Cangialosi et al. 2015
Cangialosi, Frank, Dave Levin, and Neil Spring (2015). "Ting: Measuring and Exploiting Latencies Between All Tor Nodes". In: *Proc. of the ACM Internet Measurement Conference (IMC)*, pp. 289–302.

Carvalho et al. 2018
Carvalho, Carlos, Daniel Porto, Luís Rodrigues, Manuel Bravo, and Alysson Bessani (2018). "Dynamic Adaptation of Byzantine Consensus Protocols". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. Pau, France, pp. 411–418. ISBN: 978-1-4503-5191-1. DOI: 10.1145/3167132.3167179.

Cason et al. 2021
Cason, Daniel, Enrique Fynn, Nenad Milosevic, Zarko Milosevic, Ethan Buchman, and Fernando Pedone (2021). "The design, architecture and performance of the Tendermint Blockchain Network". In: *40th Int. Symp. on Reliable Distributed Systems*. IEEE. Chicago, IL, USA: IEEE Comp. Soc., pp. 23–33.

Castro 2000 Castro, Miguel (2000). "Practical Byzantine Fault Tolerance". PhD thesis. Massachusetts Institute of Technology, Cambridge, MA,

Castro and Liskov 1999
Castro, Miguel and Barbara Liskov (1999). "Practical Byzantine fault tolerance". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 173–186.

Castro and Liskov 2001
Castro, Miguel and Barbara Liskov (2001). "Byzantine fault tolerance can be fast". In: *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN)*. IEEE, pp. 513–518.

Chandra, Griesemer, et al. 2007
Chandra, Tushar, Robert Griesemer, and Joshua Redstone (2007). "Paxos made live: an engineering perspective". In: *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 398–407.

Chandra and Toueg 1996
Chandra, Tushar and Sam Toueg (1996). "Unreliable failure detectors for reliable distributed systems". In: *Journal of the ACM (JACM)* 43.2, pp. 225–267.

Chiba et al. 2022
Chiba, Tairi, Ren Ohmura, and Junya Nakamura (2022). "Network Bandwidth Variation-Adapted State Transfer for Geo-Replicated State Machines and its Application to Dynamic Replica Replacement". In: *preprint arXiv:2204.08656*.

Civit, Gilbert, and Gramoli 2021
Civit, Pierre, Seth Gilbert, and Vincent Gramoli (2021). "Polygraph: Accountable Byzantine agreement". In: *Proc. of the 41st IEEE Int. Conference on Distributed Computing Systems (ICDCS)*, pp. 403–413.

Civit, Gilbert, Gramoli, et al. 2022
Civit, Pierre, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, and Jovan Komatovic (2022). "As easy as ABC: Optimal (A)ccountable (B)yzantine (C)onsensus is easy!" In: *Proc. of the 37th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, pp. 560–570.

Clement, Kapritsos, et al. 2009
Clement, Allen, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche (2009). "Upright Cluster Services". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. ACM, pp. 277–290.

Clement, Wong, et al. 2009
Clement, Allen, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti (2009). "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults." In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Vol. 9, pp. 153–168.

Coelho and Pedone 2018
Coelho, Paulo and Fernando Pedone (Oct. 2018). “Geographic State Machine Replication”. In: *37th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pp. 221–230. DOI: 10.1109/SRDS.2018.00034.

Corbett et al. 2013
Corbett, James C, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. (2013). “Spanner: Google’s globally distributed database”. In: *ACM Transactions on Computer Systems (TOCS)* 31.3, pp. 1–22.

Cowling et al. 2006
Cowling, James, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira (2006). “HQ replication: A hybrid quorum protocol for Byzantine fault tolerance”. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 177–190.

Crain et al. 2021
Crain, Tyler, Christopher Natoli, and Vincent Gramoli (2021). “Red Belly: A Secure, Fair and Scalable Open Blockchain”. In: *IEEE Symp. on Security and Privacy (SP)*, pp. 466–483. ISBN: 978-1-72818-934-5. DOI: 10.1109/SP40001.2021.00087.

Decouchant, Kozhaya, et al. 2022
Decouchant, Jérémie, David Kozhaya, Vincent Rahli, and Jiangshan Yu (2022). “DAMYSUS: streamlined BFT consensus leveraging trusted components”. In: EuroSys ’22. Rennes, France: Association for Computing Machinery, pp. 1–16. ISBN: 9781450391627. DOI: 10.1145/3492321.3519568. URL: https://doi.org/10.1145/3492321.3519568.

Decouchant, Kozhaya, et al. 2024
Decouchant, Jérémie, David Kozhaya, Vincent Rahli, and Jiangshan Yu (2024). “OneShot: View-Adapting Streamlined BFT Protocols with Trusted Execution Environments”. In: *International Parallel and Distributed Processing Symposium.*

Decouchant, Ozkan, et al. 2023
Decouchant, Jérémie, Burcu Kulahcioglu Ozkan, and Yanzhuo Zhou (2023). “Liveness Checking of the HotStuff Protocol Family”. In: *The 28th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, pp. 168–179. DOI: 10.1109/PRDC59308.2023.00029.

Distler 2021
Distler, Tobias (Feb. 2021). “Byzantine Fault-Tolerant State-Machine Replication from a Systems Perspective”. In: *ACM Computing Surveys (CSUR)* 54.1. ISSN: 0360-0300. DOI: 10.1145/3436728.

Distler et al. 2015
Distler, Tobias, Christian Cachin, and Rüdiger Kapitza (2015). “Resource-efficient Byzantine fault tolerance”. In: *IEEE Transactions on Computers (TC)* 65.9, pp. 2807–2819.

Dolev and Strong 1983
Dolev, Danny and H. Raymond Strong (1983). “Authenticated algorithms for Byzantine agreement”. In: *SIAM Journal on Computing* 12.4, pp. 656–666.

Domaschka et al. 2019
Domaschka, Jörg, Christian Berger, Hans P. Reiser, Philipp Eichhammer, Frank Griesinger, Jakob Pietron, Matthias Tichy, Franz J. Hauck, and Gerhard Habiger (2019). "SORRIR: A Resilient Self-Organizing Middleware for IoT Applications [Position Paper]". In: The 6th International Workshop on Middleware and Applications for the Internet of Things (M4IoT). Davis, CA, USA: Association for Computing Machinery, pp. 13–16. ISBN: 9781450370288. DOI: 10.1145/3366610.3368098. URL: https://doi.org/10.1145/3366610.3368098.

Duan et al. 2018
Duan, Sisi, Michael K. Reiter, and Haibin Zhang (2018). "BEAT: Asynchronous BFT made practical". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, pp. 2028–2041.

Dwork et al. 1988
Dwork, Cynthia, Nancy Lynch, and Larry Stockmeyer (1988). "Consensus in the presence of partial synchrony". In: *Journal of the ACM (JACM)* 35.2, pp. 288–323.

Eischer and Distler 2018
Eischer, Michael and Tobias Distler (2018). "Latency-Aware Leader Selection for Geo-Replicated Byzantine Fault-Tolerant Systems". In: *1st Workshop on Byzantine Consensus and Resilient Blockchains (BCRB '18)*, pp. 140–145. DOI: 10.1109/DSN-W.2018.00053.

Eischer and Distler 2020
Eischer, Michael and Tobias Distler (2020). "Resilient cloud-based replication with low latency". In: *Proc. of the 21st International Middleware Conference*, pp. 14–28.

Eischer and Distler 2021
Eischer, Michael and Tobias Distler (2021). "Egalitarian Byzantine fault tolerance". In: *2021 IEEE 26th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, pp. 1–10.

Eischer, Straßner, et al. 2020
Eischer, Michael, Benedikt Straßner, and Tobias Distler (2020). "Low-latency geo-replicated state machines with guaranteed writes". In: *Proc. of the 7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)*, pp. 1–9.

Faria and Correia 2019
Faria, Carlos and Miguel Correia (2019). "BlockSim: blockchain simulator". In: *IEEE Int. Conf. on Blockchain*. Washington, DC, USA: IEEE Comp. Soc., pp. 439–446.

Fischer et al. 1985
Fischer, Michael J, Nancy A Lynch, and Michael S Paterson (1985). "Impossibility of distributed consensus with one faulty process". In: *Journal of the ACM (JACM)* 32.2, pp. 374–382.

Friedman et al. 2005
Friedman, Roy, Achour Mostefaoui, and Michel Raynal (2005). "Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems". In: *IEEE Transactions on Dependable and Secure Computing (TDSC)* 2.1, pp. 46–56.

Garay and Kiayias 2020
Garay, Juan and Aggelos Kiayias (2020). "Sok: A consensus taxonomy in the blockchain era". In: *Cryptographers' track at the RSA conference*. Springer, pp. 284–318.

Garcia-Molina and Barbara 1985
Garcia-Molina, Hector and Daniel Barbara (1985). “How to assign votes in a distributed system”. In: *Journal of the ACM (JACM)* 32.4, pp. 841–860.

Gervais et al. 2016
Gervais, Arthur, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun (2016). “On the security and performance of proof of work blockchains”. In: *ACM SIGSAC CCS*. New York, NY: ACM, pp. 3–16.

Gifford 1979 Gifford, David K (1979). “Weighted voting for replicated data”. In: *Proceedings of the seventh ACM symposium on Operating systems principles*, pp. 150–162.

Gilad et al. 2017
Gilad, Yossi, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich (2017). “Algorand: Scaling Byzantine agreements for cryptocurrencies”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, pp. 51–68.

Gouveia et al. 2020
Gouveia, Paulo, João Neves, Carlos Segarra, Luca Liechti, Shady Issa, Valerio Schiavoni, and Miguel Matos (2020). “Kollaps: decentralized and dynamic topology emulation”. In: *Proc. of the 15th European Conference on Computer Systems (EuroSys)*, pp. 1–16.

Gray 1986 Gray, Jim (1986). “Why do computers stop and what can be done about it?” In: *Symposium on reliability in distributed software and database systems*. Los Angeles, CA, USA, pp. 3–12.

Guerraoui, Knežević, et al. 2010
Guerraoui, Rachid, Nikola Knežević, Vivien Quéma, and Marko Vukolić (2010). “The next 700 BFT protocols”. In: *Proceedings of the 5th European conference on Computer systems*, pp. 363–376.

Guerraoui, Pavlovic, et al. 2016
Guerraoui, Rachid, Matej Pavlovic, and Dragos-Adrian Seredinschi (2016). “Incremental consistency guarantees for replicated objects”. In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI’16)*. CONF, pp. 169–184.

Hadzilacos and Toueg 1993
Hadzilacos, Vassos and Sam Toueg (1993). “Fault-tolerant broadcasts and related problems”. In: *Distributed systems (2nd Ed.)* Pp. 97–145.

Hadzilacos and Toueg 1994
Hadzilacos, Vassos and Sam Toueg (May 1994). *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Tech. rep. TR94-1425. USA: Cornell University.

Handigol et al. 2012
Handigol, Nikhil, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown (2012). “Reproducible network experiments using container-based emulation”. In: *8th Int. Conf. on Emerging networking experiments and technologies*. New York, NY: ACM, pp. 253–264.

Herlihy and Wing 1990
Herlihy, Maurice P and Jeannette M Wing (1990). “Linearizability: A correctness condition for concurrent objects”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3, pp. 463–492. DOI: 10.1145/78969.78972.

Howard et al. 2021
Howard, Heidi, Aleksey Charapko, and Richard Mortier (2021). “Fast Flexible Paxos: Relaxing quorum intersection for Fast Paxos”. In: *Proc. of the 22nd Int. Conf. on Distributed Computing and Networking (ICDCN)*, pp. 186–190.

Hunt et al. 2010
Hunt, Patrick, 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*. Vol. 8. 9.

Jansen et al. 2022
Jansen, Rob, Jim Newsome, and Ryan Wails (2022). “Co-opting Linux Processes for High-Performance Network Simulation”. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 327–350.

F. P. Junqueira, Mao, et al. 2007
Junqueira, Flavio P., Yanhua Mao, and Keith Marzullo (2007). “Classic paxos vs. fast paxos: caveat emptor”. In: *Proceedings of USENIX Hot Topics in System Dependability (HotDep)*.

F. P. Junqueira, Reed, et al. 2011
Junqueira, Flavio P., Benjamin C Reed, and Marco Serafini (2011). “Zab: High-performance broadcast for primary-backup systems”. In: *Proceedings of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

Kapitza et al. 2012
Kapitza, Rüdiger, 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 ACM european conference on Computer Systems*, pp. 295–308.

Kirkpatrick et al. 1983
Kirkpatrick, Scott, C Daniel Gelatt Jr, and Mario P Vecchi (1983). “Optimization by Simulated Annealing”. In: *Science* 220.4598, pp. 671–680. ISSN: 0036-8075. DOI: `10.1126/science.220.4598.671`. eprint: `https://science.sciencemag.org/content/220/4598/671.full.pdf`. URL: `https://science.sciencemag.org/content/220/4598/671`.

Kogias et al. 2016
Kogias, Eleftherios Kokoris, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford (2016). “Enhancing bitcoin security and performance with strong consistency via collective signing”. In: *25th USENIX Security Symposium (USENIX Security 16)*, pp. 279–296.

Kohls and Diaz 2022
Kohls, Katharina and Claudia Diaz (2022). “VerLoc: Verifiable Localization in Decentralized Systems”. In: *Proc. of the 31st USENIX Security Symp. (USENIX Security)*, pp. 2637–2654.

Kokoris-Kogias et al. 2018
Kokoris-Kogias, Eleftherios, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford (May 2018). “OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding”. In: *2018 IEEE Symposium on Security and Privacy*, pp. 583–598. DOI: `10.1109/SP.2018.000-5`.

Köstler et al. 2023
Köstler, Johannes, Hans P. Reiser, Franz J. Hauck, and Gerhard Habiger (2023). “Fluidity: Location-Awareness in Replicated State Machines”. In: *Proceedings of the 38th Annual ACM Symposium on Applied Computing.*

Kotla et al. 2007
Kotla, Ramakrishna, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong (2007). “Zyzzyva: speculative Byzantine fault tolerance”. In: *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pp. 45–58.

Kuznetsov et al. 2021
Kuznetsov, Petr, Andrei Tonkikh, and Yan X Zhang (2021). “Revisiting optimal resilience of fast Byzantine consensus”. In: *Proc. of the 40th ACM Symp. on Principles of Distributed Computing (PODC)*, pp. 343–353.

Lamport 1977
Lamport, Leslie (1977). “Proving the correctness of multiprocess programs”. In: *IEEE transactions on software engineering* 2, pp. 125–143.

Lamport 1987
Lamport, Leslie (1987). *The writings of Leslie Lamport.* [online] `https://lamport.azurewebsites.net/pubs/pubs.html` (last accessed 09/01/2023).

Lamport, Malkhi, et al. 2010
Lamport, Leslie, Dahlia Malkhi, and Lidong Zhou (2010). “Reconfiguring a state machine”. In: *ACM SIGACT News* 41.1, pp. 63–73.

Lamport, Shostak, et al. 1982
Lamport, Leslie, Robert Shostak, and Marshall Pease (July 1982). “The Byzantine Generals Problem”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3, pp. 382–401. ISSN: 0164-0925.

Lantz et al. 2010
Lantz, Bob, Brandon Heller, and Nick McKeown (2010). “A network in a laptop: rapid prototyping for software-defined networks”. In: *9th ACM SIGCOMM Workshop on Hot Topics in Networks.* New York, NY: ACM, pp. 1–6.

B. Li et al. 2018
Li, Bijun, Nico Weichbrodt, Johannes Behl, Pierre-Louis Aublin, Tobias Distler, and Rüdiger Kapitza (2018). “Troxy: Transparent access to Byzantine fault-tolerant systems”. In: *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, pp. 59–70.

P. Li, G. Wang, Chen, Long, et al. 2020
Li, Peilun, Guosai Wang, Xiaoqi Chen, Fan Long, and Wei Xu (2020). “Gosig: a scalable and high-performance Byzantine consensus for consortium blockchains”. In: *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 223–237.

P. Li, G. Wang, Chen, and Xu 2018
Li, Peilun, Guosai Wang, Xiaoqi Chen, and Wei Xu (2018). “Gosig: Scalable Byzantine Consensus on Adversarial Wide Area Network for Blockchains”. In: *arXiv preprint arXiv:1802.01315.*

J. Liu et al. 2018
Liu, Jian, Wenting Li, Ghassan O Karame, and N Asokan (2018). “Scalable Byzantine Consensus via Hardware-assisted Secret Sharing”. In: *IEEE Transactions on Computers (TC).*

S. Liu and Vukolić 2017
Liu, Shengyun and Marko Vukolić (2017). “Leader Set Selection for Low-Latency Geo-Replicated State Machine”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.7, pp. 1933–1946.

Loo et al. 2005
Loo, Boon Thau, Tyson Condie, Joseph M Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica (2005). “Implementing declarative overlays”. In: *12th ACM SOSP*. New York, NY: ACM, pp. 75–90.

Losa et al. 2019
Losa, Giuliano, Eli Gafni, and David Mazières (2019). “Stellar Consensus by Instantiation”. In: *33rd International Symposium on Distributed Computing (DISC 2019)*.

Lu et al. 2022
Lu, Yuan, Zhenliang Lu, and Qiang Tang (2022). “Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As Pipelined BFT”. In: *Proc. of the 29th ACM Conference on Computer and Communications Security (CCS)*.

Lynch 1996 Lynch, Nancy A (1996). *Distributed algorithms*. Elsevier.

Malkhi and Reiter 1997
Malkhi, Dahlia and Michael Reiter (1997). “Byzantine quorum systems”. In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pp. 569–578.

Mao, F. P. Junqueira, et al. 2008
Mao, Yanhua, Flavio P. Junqueira, and Keith Marzullo (2008). “Mencius: Building Efficient Replicated State Machines for WANs”. In: OSDI’08. San Diego, California, pp. 369–384.

Mao, F. Junqueira, et al. 2009
Mao, Yanhua, Flavio P. Junqueira, and Keith Marzullo (2009). “Towards low latency state machine replication for uncivil wide-area networks”. In: *Workshop on Hot Topics in System Dependability*.

Martin and Alvisi 2006
Martin, Jean-Philippe and Lorenzo Alvisi (2006). “Fast Byzantine consensus”. In: *IEEE Transactions on Dependable and Secure Computing (TDSC)* 3.3, pp. 202–215.

Messadi et al. 2022
Messadi, Ines, 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 conference on 23rd ACM/IFIP International Middleware Conference*, pp. 56–68.

Miller and Jansen 2015
Miller, Andrew and Rob Jansen (2015). “Shadow-Bitcoin: Scalable Simulation via Direct Execution of Multi-Threaded Applications”. In: *8th Workshop on Cyber Security Experimentation and Test*. Berkeley, CA, USA: USENIX Association.

Miller, Xia, et al. 2016
Miller, Andrew, 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*. ACM, pp. 31–42.

Milosevic et al. 2013
Milosevic, Zarko, Martin Biely, and André Schiper (2013). “Bounded delay in Byzantine-tolerant state machine replication”. In: *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. IEEE, pp. 61–70.

Moraru et al. 2013
Moraru, Iulian, 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*. ACM, pp. 358–372.

Moraru et al. 2014
Moraru, Iulian, David G Andersen, and Michael Kaminsky (2014). “Paxos quorum leases: Fast reads without sacrificing writes”. In: *Proceedings of the ACM Symposium on Cloud Computing ((SoCC))*, pp. 1–13.

Nakamoto 2008
Nakamoto, Satoshi (2008). “Bitcoin: A peer-to-peer electronic cash system”. In: URL: `https://bitcoin.org/bitcoin.pdf` (visited on 02/14/2024).

Nayak and Abraham 2019
Nayak, Kartik and Ittai Abraham (2019). *Consensus for State Machine Replication.* [online] `https://decentralizedthoughts.github.io/2019-10-15-consensus-for-state-machine-replication/` (last accessed 10/01/2023).

Neiheiser, Matos, et al. 2021
Neiheiser, Ray, Miguel Matos, and Luís Rodrigues (2021). “Kauri: Scalable BFT consensus with pipelined tree-based dissemination and aggregation”. In: *ACM SIGOPS 28th SOSP*. Virtual Event Germany: ACM, pp. 35–48.

Neiheiser, Rech, et al. 2020
Neiheiser, Ray, Luciana Rech, Manuel Bravo, Luís Rodrigues, and Miguel Correia (2020). “Fireplug: Efficient and Robust Geo-Replication of Graph Databases”. In: *IEEE Transactions on Parallel and Distributed Systems* 31.8, pp. 1942–1953.

Nischwitz et al. 2021
Nischwitz, Martin, Marko Esche, and Florian Tschorsch (2021). “Bernoulli meets pbft: Modeling BFT protocols in the presence of dynamic failures”. In: *16th Conference on Computer Science and Intelligence Systems*. Washington, DC, USA: IEEE Comp. Soc., pp. 291–300.

Nischwitz et al. 2022
Nischwitz, Martin, Marko Esche, and Florian Tschorsch (2022). “Raising the AWAREness of BFT Protocols for Soaring Network Delays”. In: *Proc. of the 47th IEEE Conf. on Local Computer Networks (LCN)*, pp. 387–390.

Numakura et al. 2019
Numakura, Shota, Junya Nakamura, and Ren Ohmura (2019). “Evaluation and Ranking of Replica Deployments in Geographic State Machine Replication”. In: *38th International Symposium on Reliable Distributed Systems Workshops (SRDSW)*. IEEE, pp. 37–42.

Ongaro and Ousterhout 2014
Ongaro, Diego and John Ousterhout (2014). “In search of an understandable consensus algorithm”. In: *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pp. 305–319.

Pâris 1986 Pâris, Jehan-Francois (1986). “Voting with Witnesses: A Constistency Scheme for Replicated Files.” In: *ICDCS*, pp. 606–612.

Pease et al. 1980

Pease, Marshall, Robert Shostak, and Leslie Lamport (1980). "Reaching agreement in the presence of faults". In: *Journal of the ACM (JACM)* 27.2, pp. 228–234.

Ranchal-Pedrosa and Gramoli 2023

Ranchal-Pedrosa, Alejandro and Vincent Gramoli (2023). "Basilic: Resilient-Optimal Consensus Protocols with Benign and Deceitful Faults". In: *Proc. of the 36th IEEE Computer Security Foundations Symposium (CSF)*, pp. 15–30.

Riley and Henderson 2010

Riley, George F and Thomas R Henderson (2010). "The ns-3 network simulator". In: *Modeling and tools for network simulation.* Cham: Springer, pp. 15–34.

Rüsch et al. 2019a

Rüsch, Signe, Kai Bleeke, and Rüdiger Kapitza (2019a). "Bloxy: Providing Transparent and Generic BFT-Based Ordering Services for Blockchains". In: *Proceedings of the 38th IEEE Symposium on Reliable Distributed Systems (SRDS).*

Rüsch et al. 2019b

Rüsch, Signe, Kai Bleeke, and Rüdiger Kapitza (2019b). "Themis: An Efficient and Memory-Safe BFT Framework in Rust: Research Statement". In: *3rd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers.* New York, NY: ACM, pp. 9–10.

Schneider 1990

Schneider, Fred B. (1990). "Implementing fault-tolerant services using the state machine approach: A tutorial". In: *ACM Computing Surveys (CSUR)* 22.4, pp. 299–319.

Shamis et al. 2022

Shamis, Alex, Peter Pietzuch, Burcu Canakci, Miguel Castro, Cédric Fournet, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Antoine Delignat-Lavaud, Matthew Kerner, Julien Maffre, Olga Vrousgou, Christoph M. Wintersteiger, Manuel Costa, and Mark Russinovich (2022). "IA-CCF: Individual Accountability for Permissioned Ledgers". In: *Proc. of the 19th USENIX Symp. on Networked Systems Design and Implementation (NSDI).*

Sheng et al. 2021

Sheng, Peiyao, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath (2021). "BFT protocol forensics". In: *Proc. of the 28th ACM Conference on Computer and Communications Security (CCS)*, pp. 1722–1743.

D. S. Silva et al. 2021

Silva, Douglas Simoes, Rafal Graczyk, Jérémie Decouchant, Marcus Völp, and Paulo Esteves-Veríssimo (2021). "Threat adaptive Byzantine fault tolerant state-machine replication". In: *Proc. of the 40th IEEE Int. Symp. on Reliable Distributed Systems (SRDS)*, pp. 78–87.

F. Silva et al. 2020

Silva, Fábio, Ana Alonso, José Pereira, and Rui Oliveira (2020). "A Comparison of Message Exchange Patterns in BFT Protocols: (Experience Report)". In: *Distributed Applications and Interoperable Systems: 20th IFIP WG 6.1 International Conference, DAIS 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15–19, 2020, Proceedings 20.* Springer, pp. 104–120.

Singh et al. 2008
Singh, Atul, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe (2008). “BFT Protocols Under Fire.” In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation.* Vol. 8, pp. 189–204.

Song and Renesse 2008
Song, Yee Jiun and Robbert van Renesse (2008). “Bosco: One-Step Byzantine Asynchronous Consensus”. In: *Proc. of the 22nd International Symposium on Distributed Computing (DISC)*, pp. 438–450. ISBN: 978-3-540-87778-3. DOI: `10.1007/978-3-540-87779-0_30`.

Sousa and Bessani 2012
Sousa, João and Alysson Bessani (2012). “From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation”. In: *Proceedings of the 9th European Dependable Computing Conference (EDCC).* IEEE, pp. 37–48.

Sousa and Bessani 2015
Sousa, João and Alysson Bessani (2015). “Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines”. In: *Proceedings of the 34th IEEE Symposium on Reliable Distributed Systems (SRDS).* IEEE, pp. 146–155.

Sousa, Bessani, and Vukolić 2018
Sousa, João, Alysson Bessani, and Marko Vukolić (2018). “A Byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform”. In: *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, pp. 51–58.

Stathakopoulou et al. 2022
Stathakopoulou, Chrysoula, David Tudor, Matej Pavlovic, and M Vukolić (2022). “Mir-BFT: Scalable and Robust BFT for Decentralized Networks”. In: *Journal of Systems Research* 2.1.

Sui et al. 2022
Sui, Xiao, Sisi Duan, and Haibin Zhang (2022). “Marlin: Two-Phase BFT with Linearity”. In: *Proc. of the 52nd Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN).* IEEE, pp. 54–66.

Van Steen and Tanenbaum 2017
Van Steen, Maarten and Andrew S Tanenbaum (2017). *Distributed systems.* 3rd ed. distributed-systems.net.

Veronese et al. 2009
Veronese, Giuliana, Miguel Correia, Alysson Bessani, and Lau Lung (2009). “Spin one’s wheels? Byzantine fault tolerance with a spinning primary”. In: *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems (SRDS).* IEEE, pp. 135–144.

Veronese et al. 2010
Veronese, Giuliana, Miguel Correia, Alysson Bessani, and Lau Lung (2010). “EBAWA: Efficient Byzantine Agreement for Wide-Area Networks”. In: *Proc. of the 12th IEEE Int. Symp. on High Assurance Syst. Eng. (HASE)*, pp. 10–19. DOI: `10.1109/HASE.2010.19`.

Vukolić 2015 Vukolić, Marko (2015). “The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication”. In: *International Workshop on Open Problems in Network Security.* Springer, pp. 112–125.

B. Wang et al. 2020

Wang, Bozhi, Shiping Chen, Lina Yao, and Qin Wang (2020). “ChainSim: A P2P Blockchain Simulation Framework”. In: *CCF China Blockchain Conf.* Singapore: Springer, pp. 1–16.

P.-L. Wang et al. 2022

Wang, Ping-Lun, Tzu-Wei Chao, Chia-Chien Wu, and Hsu-Chun Hsiao (2022). “Tool: An Efficient and Flexible Simulator for Byzantine Fault-Tolerant Protocols”. In: *52th Annu. IEEE/IFIP Int. Conf. on Dependable Systems and Networks.* Washington, DC, USA: IEEE Comp. Soc., pp. 287–294.

Wester et al. 2009

Wester, Benjamin, James A Cowling, Edmund B Nightingale, Peter M Chen, Jason Flinn, and Barbara Liskov (2009). “Tolerating Latency in Replicated State Machines Through Client Speculation.” In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 245–260.

Wette et al. 2014

Wette, Philip, Martin Dräxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl (2014). “Maxinet: Distributed emulation of software-defined networks”. In: *IFIP Networking Conf.* Washington, DC, USA: IEEE Comp. Soc., pp. 1–9.

Winter et al. 2023

Winter, Levin N, Florena Buse, Daan De Graaf, Klaus Von Gleissenthall, and Burcu Kulahcioglu Ozkan (2023). “Randomized Testing of Byzantine Fault Tolerant Algorithms”. In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1, pp. 757–788.

Yahyaoui et al. 2024

Yahyaoui, Wassim, Jérémie Decouchant, Marcus Völp, and Joachim Bruneau-Queyreix (2024). “Tolerating Disasters with Hierarchical Consensus”. In: *IEEE International Conference on Computer Communications.* IEEE, Vancouver, Canada.

Yang et al. 2022

Yang, Lei, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse (2022). “DispersedLedger: High-Throughput Byzantine Consensus on Variable Bandwidth Networks”. In: *Proc. of the 19th USENIX Symp. on Networked Systems Design and Implementation.* NSDI’22.

Yin et al. 2018

Yin, Maofan, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham (2018). “HotStuff: BFT consensus in the lens of blockchain”. In: *arXiv preprint arXiv:1803.05069.*

Yin et al. 2019

Yin, Maofan, 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.* PODC ’19. Toronto ON, Canada: ACM, pp. 347–356. ISBN: 978-1-4503-6217-7. DOI: 10.1145/3293611.3331591.

Yu et al. 2019

Yu, Jiangshan, David Kozhaya, Jérémie Decouchant, and Paulo Esteves-Veríssimo (2019). “RepuCoin: Your Reputation Is Your Power”. In: *IEEE Transactions on Computers* 68.8, pp. 1225–1237. DOI: 10.1109/TC.2019.2900648.

Zhang et al. 2020
Zhang, Yunhao, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi (2020). "Byzantine Ordered Consensus without Byzantine Oligarchy". In: *Proc. of the 14th USENIX Symp. on Operating Systems Design and Implementation (OSDI).*

Zhao et al. 2024
Zhao, Liangrong, Jérémie Decouchant, Joseph Liu, Qinghua Lu, and Jiangshan Yu (2024). "Trusted Hardware-Assisted Leaderless Byzantine Fault Tolerance Consensus". In: *IEEE Transactions on Dependable and Secure Computing*, pp. 1–12. DOI: 10.1109/TDSC.2024.3357521.