UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA


# Threat-adaptive Byzantine Consensus

Lívio Grifo Jorge Rodrigues

**Mestrado em Engenharia Informática**

Dissertação orientada por
Prof. Doutor Alysson Neves Bessani
Prof. Doutor Vinicius Vielmo Cogo

2023

# Acknowledgments

First of all, I would like to thank my Family for their support throughout my academic path, especially my Mother, Maria, who always supported my decisions and believed in me, and my Grandfather, Francisco, for being an example of perseverance. Without them and their support, none of this could be possible.

A huge thanks to all my friends that accompanied my academic path and helped me through the years, both in my bachelor's and master's. Without them, the last years wouldn't be as enjoyable as they were. I especially want to thank Afonso Gonçalves, Bernardo Cabrita, Carlota Valério, Carolina Duque, Diogo Soares, João Leitão, João Roque, Madalena Gil, Manuel Cardoso, Pedro Silva, Rafael Abrantes, Rafael Ramires and Ricardo Golçalves.

With this said, I can only thank everyone that made this work possible. Both my advisors, Alysson Bessani and Vinicius Cogo, for all the great insights, motivational talks and especially for all the patience. Their support and availability were essential to complete this work. Thanks to Hans Reiser for his outstanding contribution to improving this work. Thanks to Christian Berger for all the hard work and commitment to guiding me, especially in the implementation and evaluation of this work. Thanks to Robin Vassantlal for his availability and help throughout this work.

Additionally, I would like to thank Faculdade de Ciências da Universidade de Lisboa for providing an excellent environment for myself and my colleagues to learn and deepen our knowledge and, for the past few years, being my second home.

Finally, a formal acknowledgement to Fundação para a Ciência e Tecnologia (FCT) funding through project ThreatAdapt (ref. FCT-FNR/002/2018) and the LASIGE Research Unit (ref. UIDB/00408/2020 and ref. UIDP/00408/2020).

*To my Mother*

# Resumo

Sistemas distribuídos são essenciais para diversos serviços e infraestruturas críticas, como sistemas financeiros, plataformas de compras "online" e redes sociais. Um sistema distribuído consiste em dispositivos autónomos que trabalham em conjunto para atingir um objetivo. Com o aumento do número de dispositivos conectados à internet, a necessidade destes sistemas tem crescido. No entanto, com o aumento da complexidade e popularidade dos serviços distribuídos, aumenta também o risco de ataques maliciosos e erros de software. Tais problemas podem levar a comportamentos arbitrários e prejudiciais conhecidos como faltas bizantinas. Para lidar com essas situações, é relevante desenvolver sistemas distribuídos que sejam confiáveis e resilientes, capazes de tolerar comportamentos imprevisíveis.

Uma abordagem comum para tolerar faltas bizantinas em sistemas distribuídos é a replicação de máquinas de estado ("State Machine Replication" — SMR), que envolve coordenar as interações dos clientes com várias réplicas independentes do servidor e manter um estado consistente em cada uma. Para as diferentes réplicas chegarem a um consenso, são necessárias várias etapas de comunicação envolvendo uma maioria de réplicas corretas (quórum). Tendo em conta o número de réplicas disponíveis, normalmente os diferentes protocolos tolerantes a faltas bizantinas ("Byzantine Fault Tolerance" — BFT), definem uma quantidade máxima de faltas que conseguem tolerar (limite de resiliência). O tamanho do quórum necessário para avançar nas diferentes etapas de comunicação depende deste número máximo e desde que o número de réplicas faltosas não ultrapassem esse valor, o protocolo funciona corretamente.

A tecnologia "Blockchain" voltou a despertar o interesse na replicação de máquinas de estado tolerante a faltas bizantinas de grande escala. Embora trabalhos recentes tenham se concentrado principalmente em melhorar a escalabilidade e débito (em termos de transações processadas por segundo) desses protocolos, poucos abordaram a latência. Para sistemas distribuídos de grande escala, uma otimização relevante é diminuir a latência observada pelos clientes.

Um dos principais fatores que pode levar a latências elevadas em sistemas de grande escala é o número elevado de réplicas que é preciso coordenar. Quanto maior o número de réplicas no sistema, maior deve ser o quórum mínimo para alcançar um consenso. Quóruns muito grandes demoram significativamente mais tempo a serem formados devido ao elevado número de mensagens que precisam ser trocadas entre as réplicas do sistema. Acrescentando a isto, um sistema em que as réplicas se encontrem muito distantes umas das outras, agrava ainda mais este problema, pois aumenta o tempo necessário para que as mensagens alcancem as réplicas necessárias.

Já foi provado que ao usar réplicas extra e dando um maior poder de voto a certas réplicas mais rápidas (replicação ponderada), é possível acelerar o sistema, criando quóruns menores. Utilizando quóruns menores, as etapas de comunicação necessárias para manter o estado consistente podem avançar com menos réplicas, acelerando assim significativamente o consenso e por sua vez as respostas aos cliente. No entanto, isto só é possível devido à utilização de réplicas extra. Se forem usados quóruns menores e não forem adicionadas réplicas extra ao sistema, o limite de resiliência do sistema diminui. Existe nesta abordagem um compromisso entre a resiliência e o desempenho.

Apresentamos FLASHCONSENSUS, uma nova transformação para otimizar a latência de protocolos de consenso BFT baseados em quóruns. O FLASHCONSENSUS usa um limite de resiliência adaptável que permite a ordenação de pedidos de clientes mais rápida quando o sistema contém poucas réplicas faltosas.

Na construção do nosso protocolo usamos como ponto de partida o BFT-SMART, WHEAT e o AWARE. O BFT-SMART é uma biblioteca de replicação de máquina de estado tolerante a faltas bizantinas de alta performance. O WHEAT adicionou ao BFT-SMART algumas otimizações como a replicação ponderada, e o AWARE melhorou o WHEAT adicionando monitorização e adaptação ao estado da rede para escolher sempre a melhor configuração de pesos para o sistema, maximizando assim as possíveis melhorias em desempenho obtidas pela utilização de replicação ponderada.

Na nossa solução, exploramos a replicação ponderada adaptativa para atribuir automaticamente um maior poder de votação às réplicas mais rápidas, formando pequenos quóruns que aceleram significativamente o consenso. Mesmo ao usar quóruns pequenos com um limite de resiliência baixo, o nosso protocolo ainda satisfaz as garantias de segurança e vivacidade padrão de replicação de máquinas de estado, graças à integração cuidadosa de técnicas de auditoria e replicação de máquinas de estado abortáveis.

Um sistema de grande escala, devido à sua quantidade de réplicas, poderá tolerar um número elevado de faltas, no entanto, o caso comum de faltas presentes no sistema costuma ser menor que o tolerável. O nosso protocolo funcionará em dois modos diferentes: um conservador, que pode tolerar o máximo de faltas, e um otimista, que irá tolerar metade das faltas originais, mas, em contrapartida, será muito mais rápido. O protocolo consegue trocar entre estes dois modos de funcionamento conforme o nível de ameaça detetado no sistema.

O protocolo começa no modo conservador e após algum tempo a funcionar corretamente passará para o modo otimista. No modo otimista, pelo limite de resiliência ser menor, é necessário detetar quando o número de faltas ultrapassa este valor. Para isto usamos uma técnica de auditoria capaz de detetar réplicas maliciosas no sistema. Se detetarmos um número maior de réplicas maliciosas do que o modo otimista tolera, o sistema aborta a sua execução, e volta a um estado consistente (mesmo que haja indícios de comportamentos maliciosos, estes não surtem o efeito desejado pelos atacantes) e a seguir muda para o modo conservador.

Resumindo, o nosso protocolo tolera o número máximo de faltas original (o número máximo

tolerável tendo em conta o número total de réplicas no sistema) e ainda consegue beneficiar das melhorias de latência obtidas com a utilização de replicação ponderada.

Para avaliar a eficácia da nossa abordagem, realizamos testes com 21 réplicas numa rede emulada semelhante às regiões da AWS (Amazon Web Services). Quisemos avaliar tanto o impacto das técnicas de auditoria adicionadas, como o desempenho geral do nosso protocolo comparativamente a outros protocolos semelhantes (por exemplo, BFT-SMaRt e AWARE).

Desenvolvemos um protótipo do FlashConsensus com base na implementação do AWARE, uma extensão do BFT-SMaRt. Tanto o BFT-SMaRt como o AWARE são bibliotecas implementadas em Java e contam com mais de 150 classes e mais de 17000 linhas de código. No final da sua implementação, o FlashConsensus aumentou estes números para mais de 170 classes e mais de 19000 linhas de código.

Ao avaliar o impacto das técnicas de auditoria, tanto em termos de latência como em taxa de transferência, foi possível observar que o impacto em termos de performance foi menor que 1%. Em termos de recursos computacionais, foi possível identificar um aumento no uso do CPU de 39%, o que é expectável devido a todas as operações criptográficas necessárias. Em termos de uso de memória, vimos um aumento de 8%.

Para avaliar o desempenho do FlashConsensus, foram feitos testes tanto para medir a latência como para medir o débito, sendo feita uma comparação com o BFT-SMaRt e com o AWARE. Em termos de latência, o FlashConsensus consegue acelerar o consenso em $3.15\times$, o que por sua vez resulta em latências para pedidos de clientes menores. Os clientes observam um tempo de resposta $1.92\times$ mais rápido comparativamente ao BFT-SMaRt e $1.39\times$ comparativamente ao AWARE. Em termos de débito, o FlashConsensus consegue ordenar 59% mais transações por segundo comparativamente ao BFT-SMaRt e 38% comparativamente ao AWARE. Os resultados obtidos mostram que o FlashConsensus é capaz de ordenar transações com finalidade em menos de $0.4s$, o que equivale à metade do tempo necessário para um protocolo semelhante ao PBFT na mesma rede, superando até mesmo o desempenho desse protocolo quando executado nas ligações de internet teoricamente mais eficientes (transmitindo a $67\%$ da velocidade da luz)

O FlashConsensus representa um avanço significativo na busca por protocolos de consenso baseados em BFT mais rápidos e eficientes. A nossa abordagem tem o potencial de melhorar substancialmente o desempenho dos protocolos de consenso BFT críticos para uma ampla gama de aplicações de sistemas distribuídos. Ao reduzir a latência, podemos tornar os sistemas baseados em BFT mais responsivos, melhorando a experiência do cliente e a eficácia geral desses sistemas.

**Palavras-chave:** Replicação de Máquinas de Estado, Tolerância a Faltas Bizantinas, Auditoria em Sistemas Tolerantes a Faltas Bizantinas, Consenso.

# Abstract

Blockchain technology has sparked renewed interest in planetary-scale Byzantine fault-tolerant (BFT) state machine replication (SMR). While recent works have mainly focused on improving the scalability and throughput of these protocols, few have addressed latency. We present FLASHCONSENSUS, a novel transformation for optimizing the latency of quorum-based BFT consensus protocols.

FLASHCONSENSUS uses an adaptive resilience threshold that enables faster transaction ordering when the system contains few faulty replicas. Our construction exploits *adaptive weighted replication* to automatically assign high voting power to the fastest replicas, forming small quorums that significantly speed up consensus. Even when using small quorums with a low resilience threshold, our protocol still satisfies the standard SMR safety and liveness guarantees, thanks to the careful integration of abortable SMR and BFT forensics techniques.

To evaluate the efficacy of our approach, we conducted experiments with 21 replicas deployed on an emulated network resembling the AWS (Amazon Web Services) regions. The results show that FLASHCONSENSUS can order transactions with finality in under 0.4s, which is half the time a PBFT-like protocol takes in the same network and even less than this protocol running on the theoretically best possible internet links (transmitting at 67% of the speed of light).

FLASHCONSENSUS represents a significant step forward in the quest for faster, more efficient BFT-based consensus protocols. Our approach has the potential to substantially improve the performance of BFT consensus protocols, which are critical to a wide range of distributed systems applications. By reducing latency, we can make BFT-based systems more responsive, improving the user experience and the overall effectiveness of these systems.



**Keywords:** State Machine Replication, Byzantine Fault Tolerance, BFT Forensics, Consensus.

# Contents

# List of Figures

# List of Tables

# Acronyms

**AWS** Amazon Web Services

**BFT** Bizantine Fault Tolerance

**CPU** Central Processing Unit

**ECDSA** Elliptic Curve Digital Signature Algorithm

**JDK** Java Development Kit

**PoC** Proof of Culpability

**RAM** Random Access Memory

**SHA** Secure Hash Algorithm

**SMR** State Machine Replication

**TLS** Transport Layer Security

**TOM** Total Order Multicast

**UML** Unified Modeling Language

**WAN** Wide Area Network

# Chapter 1

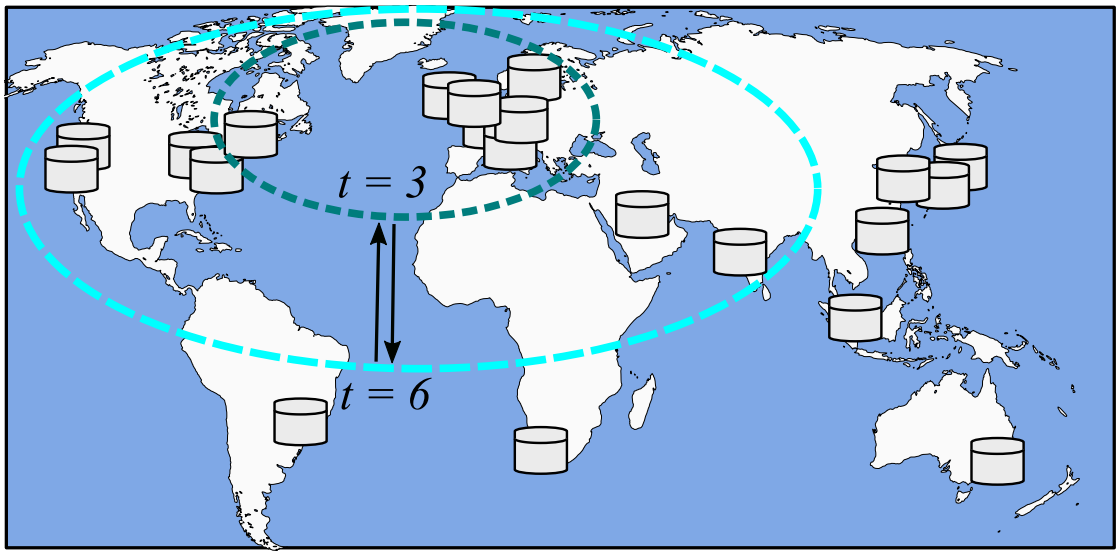
# Introduction

Distributed systems are vital in powering numerous critical services and infrastructures we rely on daily, from financial systems and e-commerce platforms to social media networks and cloud computing. A distributed system is a collection of autonomous, independent devices that work together to fulfil a goal. Such systems can be used to build distributed services and applications. With millions of users simultaneously accessing and interacting with these services from around the globe, availability, scalability, fault tolerance, and efficient resource management are essential. The need for such systems has grown over the years, especially with the increasing number of devices connected to the internet.

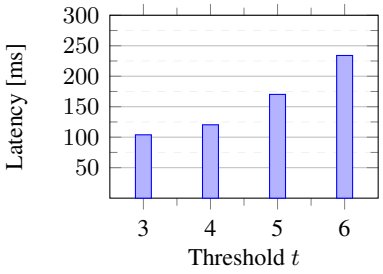
The rising popularity and importance of distributed services, such as blockchains, has increased the risk of malicious attacks. Additionally, the expanding complexity of these services increases the likelihood of software errors. Malicious attacks and software errors can cause devices in a distributed system to exhibit arbitrary and malicious behaviour, commonly referred to as Byzantine faults. Malicious participants can deviate from the expected behaviour and attempt to disrupt the system's operations by sending conflicting or erroneous information and intentionally misleading other participants. Developing a resilient distributed system that can tolerate such unpredictable behaviours, known as Byzantine faults, is highly desirable.

State machine replication (SMR) is a general approach for achieving fault tolerance in distributed systems by coordinating client interactions with a set of $n$ independent server replicas [50]. As of recently, many scalable Byzantine fault tolerant (BFT) SMR protocols have been proposed for usage in blockchain infrastructures, such as HotStuff [60], SBFT [32], Tendermint [16], Mir-BFT [56], RedBellyBC [22], Kauri [46], and Marlin [57]. These protocols employ either some dynamically elected leader [16, 32, 46, 57, 60], use multiple leaders [1, 56], or are leaderless [3, 22].

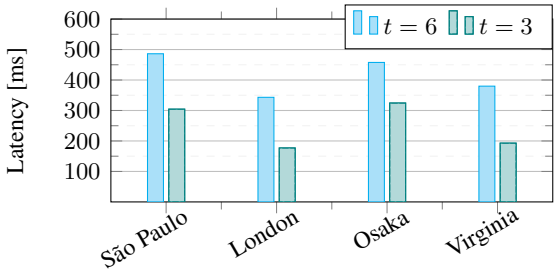
Nevertheless, consensus in all these cases requires communication steps, which must involve a majority of correct replicas to agree on a common value or decision. However, achieving consensus in the presence of malicious replicas is more challenging. Therefore, systems that operate under the Byzantine fault tolerance model assume the presence of a Byzantine adversary who can control a certain number of replicas up to a *fixed* resilience threshold denoted as $t = \lfloor \frac{n-1}{3} \rfloor$ replicas. This means the system can tolerate up to $t$ replicas behaving maliciously while achieving consensus. The system may not achieve consensus if the Byzantine adversary controls more than




(a) Weighted quorums sizes with $t = 6$ and $t = 3$.




(b) Consensus latency vs. resilience threshold.




(c) End-to-end transaction latencies observed by clients in different regions.

Figure 1.1: Weighted quorums composition and resulting BFT SMR latency for different resilience thresholds ($t$) in our $n = 21$ setup (see detains in Section 6).

$t$ replicas. Often, the quorum size for proceeding to the next protocol stage depends on this threshold, a Byzantine *t-dissemination quorum* with $\lceil \frac{n+t+1}{2} \rceil$ replicas [41]. This size equals roughly $2/3$ of the system if an *optimal* resilience threshold is used.

As for geo-replicated or planetary-scale systems, a relevant optimization goal is lowering the end-to-end latency clients observe. Sousa and Bessani [55] showed that utilizing additional spare replicas for weighted replication can accelerate the system because replicas can form smaller consensus quorums. On the downside of weighted replication, such smaller, faster quorums can exist without violating quorum intersection only due to spare replicas. Further, there is a trade-off between resilience and performance, where the smaller the size of such a fast quorum is, the more spare replicas need to exist [9].

**Smaller quorums for better latency.** To illustrate how a geo-replicated system can progress faster by accessing a proportionally smaller quorum of replicas, let us consider the scenario of Figure 1.1a, in which a weighted system [55] with $n = 21$ replicas is dispersed across all 21 AWS (Amazon Web Services) regions. If the system is configured for maximum resilience, then

it tolerates up to $t = 6$ Byzantine replicas (the highest integer satisfying $t < n/3$) and has $\Delta = 2$ spare replicas, with the size of the smallest weighted consensus quorum $Q_v^6$ containing 13 replicas (see Section 2.8 for details on these calculations). The system can instead be configured to tolerate only $t = 3$ failures, with the smallest weighted quorum $Q_v^3$ containing only 7 replicas and $\Delta = 11$. At the same time, this quorum can be composed of closer replicas that can exchange votes with each other faster, thus more swiftly proceeding through the stages of the consensus protocol (see Figure 1.1b). Finally, completing consensus faster leads to latency gains that clients around the globe can benefit from (see Figure 1.1c).

This example shows that while weighted quorums allow us to have smaller quorums that can accelerate consensus decisions, we would still need to provision some *extra* replicas to the system. By having smaller quorums, the system becomes more susceptible to the impact of faulty or compromised replicas. These extra replicas increase the total number of replicas that participate in the consensus process, thereby mitigating the influence of faulty or malicious replicas and enhancing fault tolerance by providing redundancy and increasing the chances of having a sufficient number of correct replicas in the quorum. There is a more in-depth explanation for this reasoning in Section 2.8.

## 1.1 Motivation

There have been numerous attempts to reduce the latency in SMR ([4, 9, 21, 26, 34, 42, 43, 47, 53, 55, 58]). Although most of these attempts have been successful, they come with some compromises, such as increased communication, coordination and computational overhead, more storage requirements, reliance on trusted components, a decrease in resilience to Byzantine faults and the need for specific assumptions that can be impossible to guarantee in certain scenarios. The most relevant for this work is the use of weighted replication since we want to take advantage of the ability to use smaller quorums. Weighted replication can significantly improve latency, especially in large-scale geo-replicated systems, with the expense of using more replicas without improving the fault threshold. The goal is to improve the overall latency of the system while retaining the original fault threshold without the need for additional resources.

## 1.2 Goals

In this work, our ultimate goal is to significantly reduce latency in planetary-scale BFT SMR. We want to leverage the improvements present in AWARE, such as weighted replication, and the capacity to automatically find the best weight configuration while maintaining the standard level of resilience. To achieve our goal, we need to use a technique for analyzing safety violations in BFT protocols in a novel way to detect and punish replicas that behave maliciously. By doing so, we can maintain our system running in the most optimal configuration.

The system should work on two different modes of operation, one that can withstand the maximum number of faults and one that tolerates fewer faults but is much faster. Our protocol

should be able to change between these modes of operation independently, accordingly to the threat level detected in the system.

The final system should withstand the original number of faults and still achieve low latency by using smaller weighted quorums. Using the example in Figure 1.1, we should be able to have a system that effectively tolerates $t = 6$ faults but has the speed as if it was functioning with $t = 3$, without the need to use any extra replica added to the system.

In addition, we aim to allow clients to select how many replica responses are needed to confirm that their request is complete based on their specific needs. This can further reduce observed latencies by allowing client-side speculation.

## 1.3 Contributions

In this work, we present a threshold-adaptive BFT protocol, FLASHCONSENSUS, that strives for continuous self-optimization during runtime by tuning the threshold used in consensus quorums. FLASHCONSENSUS builds on AWARE [9], an extension to BFT-SMART (a robust Java-based BFT SMR library targeting high performance) that provides automated and dynamic voting-weight tuning and leader placement for supporting the emergence of fast quorums systems with a fixed resilience threshold $t$. Our ultimate goal is to significantly reduce latency in planetary-scale BFT SMR.

In summary, we make the following contributions:

- We explore how to detect malicious behaviour under an underestimated threshold $t_{fast} < t$ by auditing the system and repairing the correct replicas state after an agreement violation happens.
- We show that it is possible to preserve the usual SMR guarantees, *linearizability* and *termination*, under the larger resilience threshold $t$, even if the agreement quorums are formed using a smaller threshold $t_{fast}$.
- We allow for client-side speculation, thus enabling a client application to minimize the observed transaction latency even further by selecting the desired consistency level.
- We conduct an extensive evaluation based on emulation of real networks to reason about possible latency gains that clients dispersed over the globe can achieve when using FLASHCONSENSUS.

The work of this thesis was presented in ConsensusDay 23 titled “Beating the Speed of Light: Low-Latency Planetary-Scale Adaptive Byzantine Consensus” [10], and is currently under submission to other conferences. A poster was also accepted in PRDC2023 titled “Faster Quorums with FLASHCONSENSUS”.

Our experimental evaluation with up to 21 replicas around the globe (more details in Chapter 6) shows that FLASHCONSENSUS can order transactions with finality in less than 0.4s, which is half

of the time required for BFT-SMART [11] in the same network. Interestingly, our latency is lower than the theoretical optimum for BFT-SMART, considering the physical location of replicas and links transmitting at 2/3 of the speed of light, which is accepted as the theoretical upper bound on data transmission speed for the internet [14, 36].

## 1.4 Structure of the Document

The rest of this document is organised as follows:

**Chapter 2** presents the background for a better understanding of this work. The chapter explains concepts such as consensus, replicas failures, and two classes of properties: Safety and Liveness. We then describe a technique to achieve those properties, known as state machine replication, and some protocols that implement it. Finally, we discuss some techniques used to improve previously mentioned protocols and describe the concept of BFT protocol forensics.

**Chapter 3** presents a few related works relevant to this thesis. The chapter discusses approaches to improve various aspects of previous protocols and their corresponding limitations.

**Chapter 4** presents the design of our protocol. We begin by introducing the system model and outlining some challenges. Next, we provide a detailed explanation of how our protocol achieves safety and liveness properties. We then describe the reconfiguration process of our protocol and explain how it functions and how we can ensure linearizability. Moreover, we discuss how our protocol can improve the overall system performance by leveraging the concept of speculation to reduce the latency of operations.

**Chapter 5** details the implementation of FLASHCONSENSUS. The chapter explains the three main components of our implementation. Firstly it describes how forensics was integrated and implemented, explaining the core classes, messages, and changes to the previous protocol. Secondly, it explains how the switch between the different modes of operation was integrated to AWARE. Lastly, it describes the Correctable interface, which provides speculation capabilities for clients.

**Chapter 6** evaluates several aspects of our implementation. Firstly this chapter describes our evaluation setup and evaluates the impact of performing forensics. Secondly, we measured our protocol throughput and latency and compared our values with other well-known protocols. Lastly, we examined how well our protocol behaves when possible faults occur.

**Chapter 7** presents our conclusions from this work.

# Chapter 2

# Background

The design of FLASHCONSENSUS incorporates ideas from several recent works on BFT. In this chapter, we review the core principles of these works. Firstly this chapter explains several communication models known as synchronous, asynchronous and partialy synchronous. Following this, the chapter also covers the core principles of consensus and replica failures and some correctness properties for distributed systems. After this, we touch upon a technique called state machine replication and some practical protocols that implemented it, specifically PBFT and BFT-SMART. Then we go through the importance of quorum sizes and why these numbers can lead to slow systems.

Two other important concepts discussed in this chapter are abortable state machine replication, a technique we used in our implementation that allows aborting the execution of client requests and change between different instances of a protocol, and the other important concept is incremental consistency guarantees, a technique that can accelerate an application by allowing speculation.

This chapter concludes by discussing two core principles for this work, weighted replication and BFT protocol forensics. Weighted replication is essential since it lays the foundation for our work. More specifically, we discuss two protocols that use this technique, WHEAT and AWARE. Lastly we discuss BFT Forensics, more specifically, forensics in PBFT and the main differences with our approach.

## 2.1 Synchrony, Asynchrony and Partial synchrony

Communication is vital in distributed systems, where multiple nodes exchange information to achieve common goals. In the context of distributed systems, we have two fundamental modes of communication: synchronous and asynchronous.

Synchronous communication refers to a scenario where communication between nodes occurs in a tightly coordinated manner, following a strict timeline. In other words, participants in a synchronous communication system must operate step by step, ensuring that each step is completed before moving to the next.

On the other hand, asynchronous communication operates without strict timing constraints. It allows nodes to operate more independently without requiring immediate responses or waiting for

specific signals.

In addition to these two models, there is another important model known as partial synchrony [24]. This model aims to find a middle ground between the synchronous and asynchronous models.

In a partially synchronous system, there are periods when the system behaves as if it were synchronous, with known bounds on message delivery times, process speeds, and other factors. However, unlike in a fully synchronous system, the partially synchronous model also allows for periods of asynchrony, where message delays and process speeds can become unpredictable. In these asynchronous periods, the system behaves like an asynchronous system, with messages potentially experiencing arbitrary delays or loss.

The concept of partial synchrony provides a more realistic assumption for many distributed systems, where there may be asynchrony moments followed by synchrony periods. By incorporating the partially synchronous model into the design and analysis of distributed systems, researchers and practitioners can develop algorithms and protocols that balance the strong guarantees of synchronous systems and the flexibility of asynchronous systems.

A popular form of *partial synchrony* is the model considering a *global stabilization time* (*GST*), where the system may initially behave asynchronously, and after some *unknown GST*, a *known* upper bound holds for all message delays.

## 2.2 Consensus

Consensus is a fundamental distributed computing problem in which a set of processes in a system needs to agree on some specific value [28]. All processes must agree on the same value, and it must have been submitted by at least one of them. Distributed systems that always respond correctly to clients despite being formed by several machines need to establish some form of consensus. The application state must be the same through all devices, and all devices should be capable of responding to clients. Consensus aims to guarantee important properties in distributed systems. Such properties are:

**Termination** Every correct process eventually decides exactly one value.

**Integrity** If a correct process decides a value $v$, then all correct processes eventually decide $v$.

**Agreement** If a correct process decides $v$, then $v$ was previously proposed by some process.

Some examples of distributed systems that need consensus are blockchains such as Bitcoin [45] and Ethereum [13], two online currencies secured without a central authority. Ethereum can also be used as a platform for creating decentralized applications.

## 2.3 Replica Failures

While trying to reach consensus, there can be several challenges. Network congestion can prevent messages from being delivered or replicas can crash, stop responding to the remaining replicas,

stop processing requests, and show arbitrary behaviour (Byzantine faults).

Byzantine failures [39] can occur for many different reasons. They can occur without human interaction or be caused by malicious users aiming to stop the system from functioning correctly. If not properly tolerated, these faults can prevent the system from making progress or influence some decisions made in the consensus. For example, some nodes participating in a consensus can provide conflicting information to different node groups, making subsets of correct nodes decide on different values (equivocation). Tolerating these types of faults is extremely important for a resilient system to continuously respond to clients and, above all, respond correctly.

An adequately designed resilient system is equipped with a predetermined threshold ($t$) that specifies the maximum number of faults it can withstand simultaneously. The system can operate correctly and maintain functionality if the number of faults remains within the established limit. However, when the threshold is surpassed, indicating that the number of faults exceeds the system's capacity for tolerance, the system can no longer ensure correct functioning.

Throughout this thesis, we will use this notation, which is summarized in Table 2.1. An important aspect is the difference between the parameter $t$ and $f$. The parameter $t$ is the commonly used parameter to define the maximum number of faults the system can tolerate. The parameter $f$ will represent the actual number of faults present in the system.

Table 2.1: Summary of notations used in this thesis.

| Symbol | Interpretation |
|---|---|
| $n$ | total number of replicas |
| $t$ | resilience threshold for the BFT protocol ($t = \lfloor \frac{n-1}{3} \rfloor$) |
| $t_{fast}$ | a faster resilience threshold ($t_{fast} = \lceil \frac{t}{2} \rceil$) |
| $f$ | actual number of faulty replicas ($f \leq t$) |

## 2.4 Safety and Liveness

Correctness is a fundamental concept in distributed systems and consensus algorithms that ensures the accuracy and integrity of system operations. Two classes of properties are used to specify correctness requirements: Safety and Liveness [38].

Liveness guarantees that something good will eventually happen, it guarantees progress, responsiveness, and the ability to complete tasks, even in the presence of failures or delays. It ensures the system remains active and continues towards its objectives. An example of a Liveness property is "all replicas eventually decide on a value".

Safety guarantees that something bad will never happen, it guarantees that all correct nodes agree on a common outcome, follow consistent rules, and prevent conflicting decisions or incorrect behavior. An example of a Safety property is "no two replicas decide on different values".

While building a distributed system, there are a number of obstacles to overcome. Failures, concurrency, synchronization, scalability and dynamic environments are some of these obstacles.

Satisfying both Liveness and Safety can be challenging. More specifically, in consensus, ensuring that decisions are made (Liveness) and made correctly (Safety) is a challenging task. Ensuring these properties in a fully asynchronous environment can even be impossible [28].

## 2.5 State Machine Replication

One way of achieving fault tolerance is through replication. State machine replication [50] is a technique used to implement fault-tolerant services by replicating the execution of requests through a set of replicas. The execution of requests in state machine replication needs to be deterministic, meaning executing the same request in different replicas results in the same state. To ensure that every replica has the same state, they need to agree on an equivalent ordering of client requests in which they are processed.

When discussing this topic, it is essential to consider two fundamental properties:

1. *Availability*: the ability to remain operational and provide services even in the face of failures or network partitions. It ensures accessibility and responsiveness to user requests.

2. *Consistency*: ensures the integrity and correctness of data across different replicas. As long as a minimum number of replicas are coordinated to reach the same state, the system can ensure correctness when responding to clients.

This approach is helpful for tolerating faults because, eventually, every correct node in the system will have the same state. Even if some replicas crash or show unusual behaviour, a client can rely on another subset of correct replicas to execute requests.

## 2.6 Byzantine State Machine Replication

**PBFT.** Castro and Liskov presented the Practical Byzantine Fault Tolerance (PBFT) [17] SMR algorithm in 1999, which became widely famous as the first practical method for tolerating Byzantine faults. If no more than $t < n/3$ replicas are faulty, PBFT provides SMR **safety** under asynchrony and **liveness** in a partially synchronous system model. Additionally, PBFT combines optimization techniques to achieve throughput comparable to a non-replicated service, being thus practical for real-world applications.

The idea of PBFT is to order requests by relying on a single leader that assigns sequence numbers to batches of requests, ensuring the state evolves consistently in all replicas throughout the system. The state of each replica includes the overall state of the service, a message log containing accepted messages exchanged between replicas, and the current view, defined by an integer. A leader $p$ in a given view $w$ is defined by $p = w \ mod \ |\mathcal{R}|$, where $\mathcal{R}$ is the set of all replicas in the system. The leader will receive client requests and multicast them to the remaining replicas.

If the leader is correct and the network and participants are well-connected, responsive, and capable of efficiently exchanging messages, PBFT's *normal case operation* is repeatedly successfully executed. The normal case operation is an agreement pattern that consists of the leader proposing a batch of operations to all replicas (PRE-PREPARE), followed by two phases of all-to-all message exchanges (PREPARE and COMMIT) in which replicas try to commit the messages (reach an agreement with the other replicas regarding the validity and ordering of the message) consistently despite Byzantine failures by the use of quorums. Quorums are sufficiently large to guarantee that any intersection of two quorums $Q_i$ and $Q_j$ contains at least one correct replica, e.g., for any pair of replicas $r_i$ and $r_j$, it holds for the quorums $Q_i$ and $Q_j$ they observe in a phase, that $|Q_i \cap Q_j| \geq t + 1$. A representation of PBFT normal case can be seen in Figure 2.1.

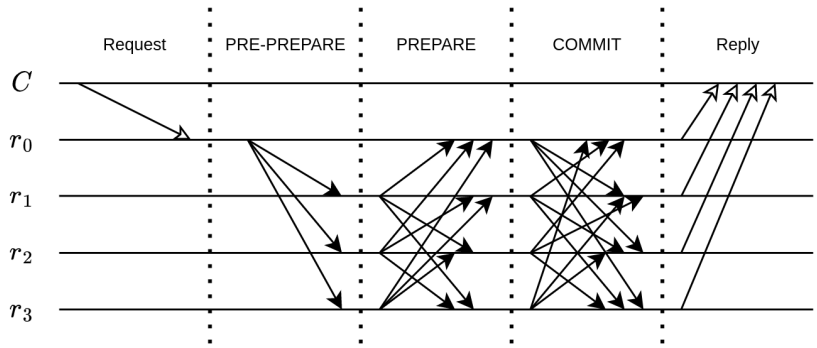



Figure 2.1: PBFT normal case operation [17].

**PRE-PREPARE**: The leader receives the client request and multicasts a pre-prepare message and appends it to its log of messages. The pre-prepare message has the format $\langle\langle \text{PRE-PREPARE}, w, s, d\rangle_{\sigma_p}, m\rangle$, where $w$ is the current view, $s$ is a sequence number attributed to the request by the leader, $d$ is the message digest $D(m)$, $m$ is the request message, and $\sigma_p$ is a signature done by the leader to prove to other replicas, that the leader sent the message. These pre-prepare messages are used to prove that the request was assigned to the sequence number $s$ in view $w$ in possible view changes. A replica that receives a pre-prepare message accepts it if the signature is correct, and if receiving replica is in view $w$ and has not accepted a pre-prepare message for view $w$ and sequence number $s$ containing different digests.

**PREPARE**: After accepting a pre-prepare message, a backup replica multicasts $\langle PREPARE, w, s, d, b\rangle_{\sigma_b}$, where, $b$ is the backup identifier, and $\sigma_b$ is a signature that proves that replica $b$ sent the message. The protocol does not send the whole message since the digest is enough to prove that the previously received $m$ in the pre-prepare corresponds to this prepare. Replicas accept prepare messages if valid and add them to their logs. A prepare message is valid if their signature is correct and their view number is the same as the receiving replica view number. There is a predicate defined as $prepared(m, w, s, b)$ that evaluates as true if a replica $b$ has inserted the request $m$ in its log, received a pre-prepare message for $m$ in view $w$ with sequence number $s$, and has at least $2t$ prepares from different backups that match the pre-prepare.

**COMMIT**: When the predicate $prepared(m, w, s, b)$ is $true$, replica $b$ multicasts a

$\langle COMMIT, w, s, D(m), b\rangle_{\sigma_b}$. When a replica receives a valid commit message, it appends it to its log. For this phase, there are two important predicates. Predicate $committed(m, w, s)$ is true if $prepared(m, w, s, b)$ is true for all replicas $b$ in a set of $t + 1$ non-faulty replicas. Predicate $committed\text{-}local(m, w, s, b)$ is $true$ if $prepared(m, w, s, b)$ is $true$ and $b$ has accepted at least $2t + 1$ commits. After $committed\text{-}local(m, w, s, b)$ is $true$, replica $b$ executes the operation requested in $m$ and sends a reply to the appropriate client (sender of $m$).

The problem of a faulty leader is solved by the *view change* sub-protocol if sufficient $(t + 1)$ replicas support the change of leader. Replicas can detect a faulty leader using timeouts, preventing backup replicas from waiting indefinitely for a batch to be proposed. During a view change, the newly elected leader collects the current status from a quorum of replicas and defines consistent decisions for pending instances.

**BFT-SMaRt.** BFT-SMaRt [11] is a robust Java-based BFT SMR library that targets both high performance in fault-free executions and correctness if faulty replicas exhibit arbitrary behaviour. Although this library was designed with Byzantine faults as the main focus, it can also be set up only to tolerate crash faults, reducing the number of replicas needed to achieve consensus.

BFT-SMaRt works just like PBFT, running a three-phase (here called Propose, Write and Accept) *consensus instance* for defining the batch of transactions to be processed, as illustrated in Figure 2.2. In contrast to PBFT, clients send requests to all replicas. Following the completion of total ordering and execution of the request, as in the case of PBFT, all replicas respond to the client. The client then collects a minimum of $t + 1$ matching responses and verifies the validity of the received value. The analogous of a view change in BFT-SMaRt is its synchronization phase, where a new leader is responsible for concluding pending consensus instances and starting instances for future slots.

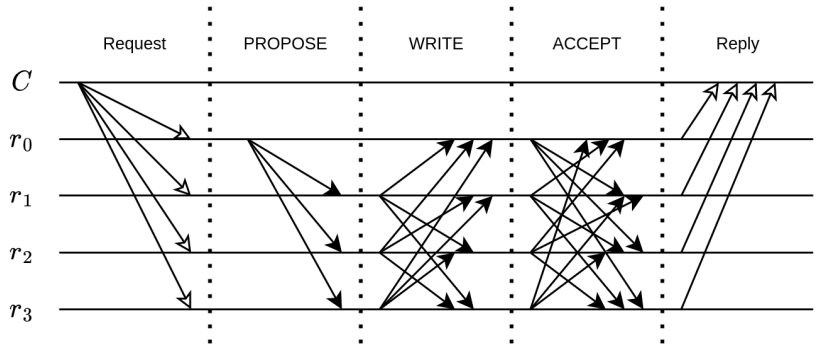



Figure 2.2: BFT-SMaRt normal case operation [11].

## 2.7 Quorum Size

As mentioned in Section 2.6, to tolerate faults through consensus, a majority of replicas need to agree. The replicas belonging to this majority compose what is called a quorum. The number of required replicas in said quorum is defined by the protocol implementing the consensus itself, as

long as it ensures that the system can agree on the same value. The number of replicas needed also depends on the number of faults the system wants to tolerate. The higher the number of faults, the higher the number of replicas required for the quorum. In PBFT, the total number of replicas in the system must be $n \geq 3t+1$. Meaning that if a system is configured to tolerate one fault ($t = 1$), its total number of replicas must be four ($n = 4$).

Suppose a system needs to have a majority of replicas agreeing on some value, ensuring at the same time that there is not any group of replicas deciding upon two different values due to some Byzantine replica. In that case, the system needs to have a quorum of $2t + 1$ replicas to achieve consensus. Using $2t + 1$ replicas ensures that if there are two quorums, each with $2t + 1$ replicas, each quorum intersects in at least $t+1$ replicas. At the quorum intersection, as long as the number of faulty replicas never surpasses $t$, there will always be a non-faulty replica that does not choose two different values. Considering this, if one quorum of $2t + 1$ replicas decides on value $v$, and the other decides on value $v'$, their intersection (containing $t+1$ replicas) must decide on the same value, meaning that $v = v'$. This rationale ensures that two quorums will never decide upon two different values.

This quorum size can be problematic in systems that require fast responses. Responding to a client requires the system to wait for the exchange of many messages between replicas in order to achieve consensus. The higher the number of faults tolerated, the higher the total replicas in the system. A high number of replicas can pose a significant latency slowdown, especially in a geographically distributed system. The higher the number of replicas and the further away they are, the worse the latency will be.

## 2.8 Weighted BFT Replication

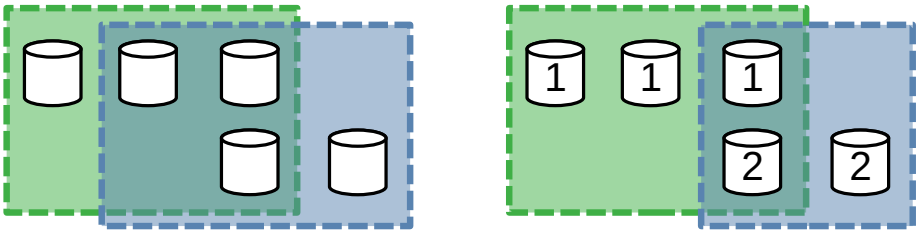
WHeight-Enabled Active replicaTion (WHEAT) [55] is an empirical design for optimizing BFT SMR for geographically dispersed deployments. A core insight of WHEAT is that client latency can be reduced by utilizing $\Delta$ additional replicas that do not contribute to increasing the resilience threshold, i.e., $n \geq 3t + 1 + \Delta$.

Quorums in WHEAT are not formed using an egalitarian Byzantine majority of replicas like in other BFT works (e.g., [17, 54, 57, 60]), but are formed instead by weighted replication, which enables proportionally smaller quorum sizes. The key lies in selecting a group of well-connected replicas, referred to as a clique, to form smaller quorums. These replicas have low latency among themselves, enabling them to coordinate and make consensus decisions efficiently (see Figure 1.1a). The approach still ensures the availability of the replicated system since other replicas with lower voting weight can be accessed to form fallback quorums.

BFT systems typically probe a Byzantine dissemination quorum, which contains $\lceil \frac{n+t+1}{2} \rceil$ replicas [41] as shown in Figure 2.3a. With $n = 5$ replicas and $t = 1$, the quorum size needs to be $4$ to ensure these quorums intersect in at least $t + 1$ replicas.

Albeit, it is also possible to satisfy the *same intersection property* by relying on specific replicas more than on others, which is captured by assigning *weights* as we can see in Figure 2.3b. A

fast quorum comprises three replicas with a total voting weight of five. This quorum is smaller than an egalitarian quorum, yet it guarantees the intersection since it contains all the replicas with high voting power.




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

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

Figure 2.3: BFT quorums for $n = 5$, $t = 1$, and $\Delta = 1$ (on b).

Now, suppose that the fastest, geographically nearest replicas are the ones in this particular quorum. They can progress the voting phases of consensus more swiftly, as they need to wait less time to collect the necessary votes. By accelerating consensus, WHEAT can also decrease the overall latency of the system [55]. Unlike traditional systems like PBFT, WHEAT does not wait for a predefined number of replicas but waits for a predefined sum of votes of $Q_v$.

To distribute weights (or voting power) among the replicas, WHEAT uses a bimodal scheme that divides the voting power in such a way that the $2t$ best-connected replicas have a voting power of $V_{max} = 1 + \frac{\Delta}{t}$ while the remaining replicas have a voting power of 1. As a result, the number of votes required for a quorum is $Q_v = 2t \cdot V_{max} + 1$. This distribution ensures that even if $t$ best-connected replicas fail, it is possible to form quorums using the other replicas.

Notice that the size of the *smallest* quorum only depends on the chosen threshold $t$, not on the actual size of the system $n$. When using this weighting scheme, all quorums contain between $2t + 1$ and $n - t$ replicas.

Considering a BFT system that tolerates one fault with an extra replica, $n = 5$, $\Delta = 1$ and $t = 1$, its weight distribution can be seen in Figure 2.3b. The fastest quorum comprises three replicas that equate to five votes, the minimum required to ensure that all quorums intersect in at least $t + 1$ replicas in this scenario. The quorum size, in this case, is smaller than the standard $2t + 1$, resulting in a reduced number of replicas required for consensus, ultimately leading to a decrease in latency for the system.

A significant setback from WHEAT is that the latency improvement can vary depending on the chosen weight configuration. A poorly chosen configuration limits this latency improvement, despite using more resources such as the extra replicas. In addition, if there are significant connection changes between replicas, an administrator must also choose a new configuration manually to reflect these changes.

**AWARE.** Distributing voting weights is difficult in practice, as the decision of what is the optimal weight configuration for a given set of network characteristics is non-trivial. To tackle this

issue, AWARE (Adaptive Wide-Area REplication) [9] allows geo-replicated state machines to self-optimize at runtime. As an automated dynamic weight tuning and leader placement scheme, AWARE supports the emergence of fast quorums in the system.

At its core, AWARE monitors latencies between all replicas and creates a corresponding latency matrix. This matrix is used to predict the expected consensus latency for several configurations of weight distributions and leader locations. Given a fixed threshold $t$, AWARE automatically chooses the fastest configuration found using the latency matrix and a meta-heuristic called simulated annealing [35]. By running this method periodically, AWARE can adapt to changes in the network: when the latency matrix is recalculated and changes are detected, the system automatically reconfigures the weight distribution and/or leader location to better suit the current network conditions. This reconfiguration can be made without administrator's interference, unlike WHEAT.

## 2.9 Abortable State Machine Replication

Abstract [5] is a design methodology to simplify the development and reconfiguration of *abortable* replicated state machines, which are, in contrast to traditional replicated state machines, allowed to abort the execution of client requests, and then re-direct this responsibility to another Abstract instance.

The design is flexible since the decision of what might be the "common case", i.e., the criteria for an optimization (such as speculating on the absence of faulty replicas or request contention) is left to the system designer. The only aspect that needs to be ensured is the safety of operations while liveness is ensured by eventually falling back to a backup instance that never aborts, e.g., PBFT. Novel Abstract instances can be composed by linking existing ones, i.e., specifying which one takes over if the other aborts. This method is implemented by a *switching mechanism* that glues together the different Abstract instances. To make this work, each Abstract instance – except the backup – needs to implement an abort subprotocol that is exposed to the next Abstract instance. In this way, the next Abstract instance can be safely initialized by utilizing the abort indications of the last, aborting Abstract instance.

## 2.10 Incremental Consistency Guarantees

*Correctables* [31] are a programming abstraction that allow client applications to work with incremental consistency guarantees. This can accelerate the application by allowing it to speculate with intermediate results. The need to form quorums to maintain consistency makes the application only respond to a client after a quorum with sufficient replicas is formed. The higher the number of replicas needed to form quorums, the slower the response is delivered to the client. Choosing the desired consistency allows a client to receive a much faster response in exchange for a lower level of consistency. For instance, a client can wait for $t+1$ replicas instead of the standard $2t+1$, exchanging some consistency for a faster response time. The client can even wait for only one

replica if desired, having the fastest response possible but having no consistency guarantees.

Correctables only have three states: `updating`, `final` and `error` state. The `updating` state allows a client application to access the intermediate result along with its respective consistency level, while the `onFinal` callback informs the application that an operation successfully completed (see Figure 2.4).

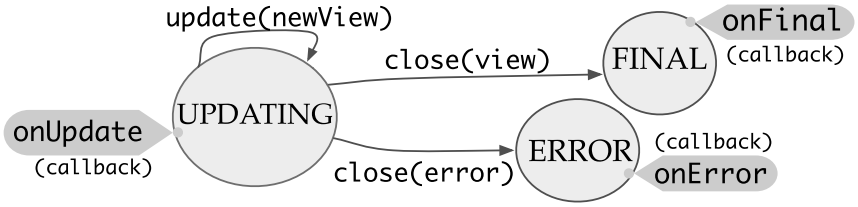



Figure 2.4: The correctable programming model [31].

## 2.11 BFT Protocol Forensics

Most BFT works focus on preventing something bad from happening, i.e., preserving safety. BFT protocol forensics [51] focuses on the day after (i.e., on events after an erroneous behaviour happened in a system), detecting malicious behaviour and identifying which replicas did not behave according to the protocol. A successful BFT protocol forensics identifies as many malicious replicas as possible with irrefutable cryptographic *proofs of culpability* (PoC). BFT forensics used a specific notation to identify some crucial parameters.

BFT forensics considers forensic support for multiple BFT protocols. The most important one for this work is PBFT-PK, a variant of PBFT in which all messages are signed. At the end of PREPARE and COMMIT phases, replicas create a message certificate with $2t + 1$ signed messages to prove which replicas were used to prepare or commit a given value in a specific view. Using these certificates, it is possible to, without any doubt, prove that a given replica participated in the agreement phase to vote on a specific value in a specific view.

Each replica sends a reply to the client after completing the agreement. If a client detects conflicting replies, it can use the message certificates from PREPARE and COMMIT phases to pinpoint and blame the responsible replicas for generating the conflicting values. This is done by identifying the Byzantine replicas at the intersection of the aggregate signatures justifying the conflicting messages. More specifically, if conflicting values are detected, at least two quorums agreed on different values. Since a quorum is formed by $2t + 1$ replicas, at least the intersection of both quorums ($t + 1$ replicas) voted for conflicting values. These votes can be used to prove that the replicas at the intersection are malicious.

PBFT-PK forensic support can, given the actual number of faults in the system is not greater than $2t$, detect and identify at least $t + 1$ faulty replicas. It is impossible to perform forensics if there are more than $2t$ faults. This limitation will be considered when developing our solution. Further, we only need a single transcript from one honest replica to obtain the *proof of culpability*. These transcripts are the set of aggregate signatures created during protocol execution.

## 2.12 Final Remarks

This chapter introduced some essential concepts to understand the background related to fault-tolerant distributed systems and the reasoning for our approach. Consensus and replicas failures were the first concepts explained in this chapter and are essential to understand the ultimate goal of a fault-tolerant distributed system.

We explored two desirable properties in distributed systems and examined various techniques that can be used to achieve these properties. We also examined two protocols that utilize these techniques and practically achieve both safety and liveness, which are essential for ensuring the proper functioning of a distributed system.

In addition to these protocols, we discussed one possible optimization to improve latency, known as weighted replication and two techniques that can be used in BFT systems to detect faulty replicas and to allow clients to trade consistency for speed when receiving replies from a BFT system.

All previously mentioned concepts and optimizations discussed will collaborate to establish the groundwork for our research. Based on the context provided, the upcoming chapter will delve into various works related to the field of Byzantine tolerance that are relevant to our work.

# Chapter 3

# Related Work

Many works propose optimizations for BFT SMR considering the adaptation to changing conditions, the geographical distribution of replicas, and low latency. In this chapter, we discuss some of these works with a few additional related works on BFT forensics.

## 3.1 Adaptivity in BFT SMR

Making BFT protocols adaptive to their environment has been studied in multiple works [6, 7, 15, 18, 25, 40, 48, 52]. RBFT [6] monitors system performance under redundant leaders to assert that a faulty leader can not purposely degrade performance. Optimizing the leader selection has been studied in several works (e.g., [25, 40]). Further works investigated adaptively switching the consensus algorithm [7, 15], strengthening the protocol by reacting to perceived threat level changes [52], being network-agnostic (tolerating a higher threshold in synchronous networks) [12] or adapting the state transfer strategy to the available network bandwidth [18].

Focusing on redundant leaders may increase computational overhead and communication costs. Optimizing leader selection may also have a limited impact if the remaining replicas are unreliable.

There is an opportunity involving the incorporation of adaptability in system configurations, such as leader and replica weights, as well as the threshold, without compromising the system's resilience. This can be achieved by integrating BFT forensics, enabling the system to dynamically adjust and optimize its parameters while maintaining robustness against Byzantine faults.

## 3.2 Geographically dispersed SMR

Various works studied the improvement of SMR for WANs [2, 21, 26, 27, 42, 43, 47, 49, 58, 59]. One of the first of these works is Mencius [42], which optimizes performance in WANs using a rotating leader scheme that allows clients to pick their geographically closest replica as its leader. Differently from Mencius, which tolerates only crash faults, EBAWA [58] uses the same rotating leader technique together with trusted components on each replica to tolerate a minority of Byzantine faulty replicas in a protocol that requires the same number of communication steps as

Mencius. Steward [2] proposes a hierarchical, two-layered architecture for replication. Regional groups within a system site run Byzantine agreement, and these replication groups are finally connected over a CFT protocol. Stewards' idea of a hierarchical architecture was later employed in the recent Fireplug [47] for efficient geo-replication of graph databases by compositing multiple BFT-SMaRt groups. PBFT-CS refines PBFT using client-side speculation. Clients send subsequent requests after predicting a response to an earlier request without waiting for the earlier request to commit – however clients need to track and propagate the dependencies between requests [59]. WHEAT optimizes BFT SMR latency by incorporating weighted replication and tentative executions [55], while AWARE enriches WHEAT through self-monitoring capabilities and dynamic optimization, by adjusting weights and leader position [9].

Mencius's rotating leader scheme may increase overhead due to frequent leader changes. Similarly, EBAWA's reliance on trusted components may increase the protocol's complexity and limit scalability. The hierarchical architecture proposed by Steward may also introduce additional communication overhead and coordination challenges, especially in large-scale systems. While Fireplug addresses some of these challenges, its effectiveness may be limited in highly dynamic environments. PBFT-CS's client-side speculation may result in increased communication overhead due to the need to accurately track and propagate dependencies between requests. WHEAT's approach to optimizing BFT SMR latency through weighted replication and tentative executions may reduce resilience to Byzantine faults. Additionally, while AWARE enriches WHEAT with self-monitoring capabilities and dynamic optimization, it suffers from the same limitation.

By employing BFT forensics, it can be possible to use smaller consensus quorums safely, thus mastering the resilience-performance trade-off that limited AWARE performance and achieving a significant decrease in consensus latency.

## 3.3 Fast BFT

It has been shown that having additional redundancy (and using a less-than-optimal resilience threshold) can be efficiently utilized to develop faster consensus variants, i.e., two-step Byzantine consensus [34, 37, 44], or even one-step asynchronous Byzantine consensus for scenarios that are contention-free [29, 53]. DuoBFT [4] uses the hybrid and Byzantine fault models where clients can choose the favored model for each command. Since hybrid commits take fewer communication steps and use smaller quorums than BFT commits, clients benefit from low-latency commits in the hybrid model.

While having additional redundancy can lead to faster consensus, it also means more overhead in terms of communication and storage requirements. Additionally, the faster consensus variants may not be as fault-tolerant as the traditional Byzantine fault-tolerant consensus protocols, and their resilience may depend on the specific assumptions and conditions of the scenario. Furthermore, the effectiveness and efficiency of the hybrid models, such as DuoBFT, may depend on the availability and reliability of the Byzantine fault model and the hybrid consensus mechanisms.

It is desirable to guarantee safety and liveness for the optimal resilience threshold (roughly 2/3

of the system) and always assume a Byzantine adversary.

## 3.4 BFT Forensics

BFT forensics is a technique for analyzing safety violations in BFT protocols after they happened [51], yielding results such as that at least $t + 1$ culprits can be identified in case of an equivocation (with the accountability of up to $2n/3$ replicas that may actually be Byzantine). Polygraph [19] is an accountable Byzantine consensus algorithm tailored for blockchain applications that allow the punishment of culprits (e.g., via stake slashing) in case of equivocations. A simple transformation to obtain an accountable Byzantine consensus protocol from any Byzantine consensus protocol has been proposed in [20], introducing an overhead of two all-to-all communication rounds and $O(n^2)$ exchanged bits of information in all executions with up to $t$ faulty processes in the common case.

Even though all these works provide advancements in accountability, they also have certain limitations. Forensics techniques often require significant computational resources and cryptographic operations to gather and analyze evidence of malicious behaviour. Additionally, the communication overhead associated with collecting and verifying cryptographic evidence from all replicas can impact the scalability and efficiency of the system.

Employing BFT forensics for incident discovery can allow for underestimating the resilience threshold and employing smaller quorums.

## 3.5 Final Remarks

In this chapter, we overviewed works done on BFT SMR optimization. We were able to detect some current limitations.

Using hierarchical architectures can introduce additional communication overhead and coordination challenges. Using redundant or rotating leaders schemes can increase computational overhead, and reliance on trusted components can increase a protocol's complexity. Client-side speculation may also increase communication overhead, and using weighted replication may reduce resilience. Having additional redundancy can lead to faster consensus, but it means higher costs and more overhead in terms of communication and storage requirements. Faster consensus variants may also not be as fault-tolerant as the traditional Byzantine fault-tolerant protocols, and their resilience may depend on specific assumptions and conditions.

Taking these limitations into consideration, in the next chapter, we will present our proposed solution, which tries to improve upon previous works without the abovementioned limitations.

# Chapter 4

# Threshold-Adaptive BFT: FLASHCONSENSUS

Considering the limitations and prior research (see Chapter 3), this thesis aims to present a solution to decrease overhead and resources while maintaining the performance improvements of previous works.

FLASHCONSENSUS provides continuous self-optimization at runtime by adapting the resilience threshold and changing weights to enable the emergence of smaller consensus quorums for low-latency transaction execution. To achieve this, it aims for the best of both worlds since it maintains the maximum resilience of the system for supporting diagnosis and repairs while continuously attempting to run consensus instances faster by optimizing the system for the expected common case with few or no failures.

This dual approach is implemented using abortable state machines, as previously done for performance [5] and resource efficiency [23]. The system starts in a *conservative mode* by running AWARE tolerating $t$ failures. If nothing bad happens for a certain number of consensus instances, the system switches to a more optimistic *fast mode* that tolerates only $t_{fast} < t$ failures. While the leader is correct and the number of actual failures $f$ does not surpass $t_{fast}$, FLASHCONSENSUS stays in this configuration and uses the flexibility of weighted quorums to accelerate the termination of consensus. If latency gains do not match expectations, the leader is found to be faulty, or if correct replicas detect equivocations, FLASHCONSENSUS switches back to its most resilient configuration. Figure 4.1 illustrates the two modes of operation.

Even building upon the features of AWARE [9], which can reconfigure weights based on the latency map of the system, this design encompasses several challenges. First and foremost, we need robust mechanisms for diagnosing the system when there are more than $f > t_{fast}$ failures. Second, we need a robust reconfiguration mechanism to safely abort the fast configuration and move to the conservative one in such situations. Finally, the client-replica contract needs to be refined for clients to tune the system for low latency or consistency under failures, depending on the application's needs. These issues are discussed in the following subsections.

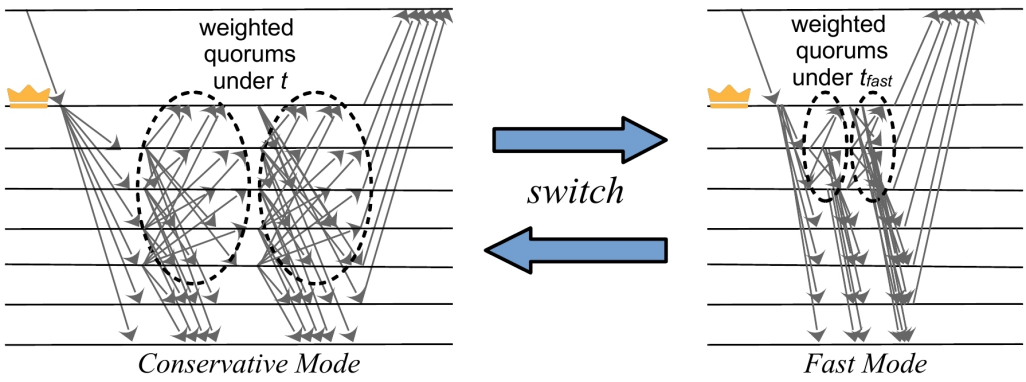



Figure 4.1: FLASHCONSENSUS two modes of operation.

## 4.1 System Model

We consider the same system model used in BFT-SMART [11] and in several important BFT protocols such as HotStuff [60] and PBFT [17]. In our system, there is a set of $n$ replicas, $\mathcal{R} = \{r_0, .., r_{n-1}\}$, and an arbitrarily large set of clients $\mathcal{C}$.

**Network Model.** Communication between all nodes in the system is done through point-to-point channels that are authenticated and reliable. Reliable communication channels can be built on top of unreliable channels, as long as they are fair-loss, i.e., eventually deliver messages if retransmission attempts are used. Further, we assume a weak form of *partial synchrony* as explained in 2.1.

**Fault Model.** We assume there are at most $t < n/3$ Byzantine replicas and an unbounded number of Byzantine clients. Byzantine processes may behave arbitrarily and even collude under the orchestration of an adversary, but they are still limited in their computational capabilities, i.e., they can not break our cryptographic primitives.

**Cryptographic Primitives.** We assume a trusted setup in which public key material is distributed among the processes. Every replica $r$ possesses secret and public keys $(SK_r, PK_r)$, and all public keys are known to all replicas and clients. Additionally, replicas are equipped with a `sign(·)` and `verify(·)` primitive to sign and verify messages. These are used to create message certificates and PoCs that play an important role in BFT forensics. Finally, we assume a collision-resistant hash function `hash(·)`.

## 4.2 Challenges and Big Picture

The main problem when speculatively using a lower resilience bound $t_{fast} < t$ for consensus is that a Byzantine attacker that controls $f$ replicas (such that $t_{fast} < f \leq t$) can cause equivocations. In this case, two correct replicas can be convinced to decide different batches of transactions for

the same consensus instance, as quorums for the lower threshold $t_{fast}$ do not necessarily overlap in at least one correct replica.

A key insight of our work is that we can utilize *BFT protocol forensics* [51] in a novel way, not as a forensic tool to investigate the "day after" but rather as a protective countermeasure against Byzantine attackers. In BFT protocol forensics, the client is the one who detects conflicting values based on replicas' logged signed messages and pinpoints equivocating replicas. In our solution, the responsibility of detecting faulty replicas is imposed on all correct replicas, so that the system can autonomously detect and expel equivocating parties. In a system tolerating up to $t_{fast}$ faulty replicas, audits can detect agreement violations and identify up to $t_{fast} + 1$ faulty replicas [51]. Using $t_{fast} = \lceil \frac{t}{2} \rceil$ guarantees that audits are always supported for up to $t$ faulty replicas. The system can recover from these violations by purging the detected violators from the system and rolling back the divergent decisions of correct replicas to a consistent state. This continuous auditing is important not only as a recovery mechanism but as a *deterrent to attacks* since perpetrators know they will be identified and expelled from the system.

The fact that a decision can be rolled back on replicas may lead transaction outcomes observed by clients to be undone, affecting the finality/durability of such transactions. Therefore, we have to modify the matching replies requirements on clients to ensure they can know when an operation *completes* in the system, ensuring linearizability as in standard SMR [17]. Another important contribution of our work is deriving the exact number of matching replies a client needs to expect to preserve linearizability even when consensus agreement violations are possible.

Minimizing consensus latency, but then letting clients wait for *more-than-usual* replies from all over the world to preserve linearizability counteracts our goal of reducing the end-to-end request latency. For this reason, we extend the programming model of BFT SMR with *correctables* that allow client applications to use incremental consistency guarantees [31], which simplifies and abstracts client-side speculation. We show that by using this abstraction, clients can lower their transaction latency even further.

Lastly, we need to take *liveness* into special consideration. SMR liveness requires requests issued by correct clients to be eventually completed. This property can be endangered if the protocol operates with a lower threshold $t_{fast} < t$ and there are $f > t_{fast}$ Byzantine replicas that stay silent, i.e., do not reply to the client or participate in consensus quorums. To avoid this scenario, we reuse the idea of *abortable state machine replication* [5]. If the system blocks, we abort the execution of the fast mode of FLASHCONSENSUS and start a more resilient protocol instance, which uses the maximum resilience threshold $t$.

## 4.3 Dealing with $f > t_{fast}$ Failures

The key challenge in devising FLASHCONSENSUS is ensuring SMR safety and liveness when the system is in fast mode and $f > t_{fast}$. In the following, we detail how we detect replicas misbehaviour affecting the safety and liveness of the system.

### 4.3.1 Safety

When running in fast mode, the adversary can control more than $t_{fast}$ replicas and cause equivocations in the system. This situation might lead correct replicas to decide different transaction batches in a consensus instance since fast mode's smaller quorums are not guaranteed to overlap in at least one correct replica.

To limit this scenario, the system must detect if the actual number of faults ($f$) surpassed the resilience threshold of the fast mode ($t_{fast}$) and, if so, revert the system to its more resilient configuration. For detecting faults, we need to check the state of the replicas periodically to ensure they are consistent, which is done through *checkpoint messages*, as in PBFT [17]. After every $k$ completed consensus instances, each replica takes a snapshot of the service state (as already done in BFT-SMART and AWARE), and broadcasts a signed message containing the hash of this snapshot and its last executed consensus number to all replicas. Every replica waits for $n - t$ matching checkpoint hashes for the same consensus number to define the checkpoint as *stable*. If during this process, a correct replica (the auditor) detects non-matching checkpoints, it runs the lightweight forensics procedure of Figure 4.2 to identify and obtain a non-repudiable *Proof-of-Culpability* (PoC) for the protocol violators.

**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 the $t_{fast} + 1$ replicas that provided signed ACCEPT messages for both decisions. These two conflicting proofs are the PoC for the $t_{fast} + 1$ misbehaving replicas.

Figure 4.2: Lightweight forensics procedure.

In BFT protocol forensics, the client is responsible for detecting conflicting values and requests replicas for proof of culpability (PoC), detecting which replicas are faulty. In our solution, the responsibility of detecting faulty replicas is moved to the replicas. After running the forensics procedure, if a PoC is produced for one or more replicas, the auditing replica broadcasts this PoC to all replicas making the system switch to the conservative mode and expel the misbehaving replicas, as described in Section 4.3.4.

Another way the forensics protocol might be triggered is if a client detects non-matching signed replies for one of its transactions. When this happens, the client sends a *panic message* with the collected conflicting replies to the replicas. A replica that receives one of these messages also triggers the forensics protocol.

### 4.3.2 Liveness

Besides equivocations, $f > t_{fast}$ adversary-controlled replicas can stay silent and negatively affect the liveness of the system. More specifically, in such situations, there will be less than $n - t_{fast}$ correct replicas in the system, subverting a key liveness assumption of the consensus protocol operating with no more than $t_{fast}$ failures. This case can lead to two unfavorable situations. First, client requests might not be ordered causing a timeout to be triggered, which makes the system progress to a leader change (i.e., BFT-SMART's synchronization phase). As it will be explained in next section, this sub-protocol reverts the system to the conservative configuration in which up to $t$ failures are tolerated.

Second, requests might be ordered but faulty replicas might not send replies to the client, preventing the client from consolidating the request result. In this case, the client could send a *panic* message to the replicas asking it to switch to the conservative mode. Replicas might trigger a leader change to switch the system's mode. However, it must be done carefully since a malicious client can easily create such a message to disallow the system to operate in fast mode. This type of weakness is inherent to many optimistic protocols [5]. Since the use of this mechanism can make FLASHCONSENSUS very fragile, as a single malicious client can undermine our latency improvements, we propose an alternative approach. If the client does not receive the required number of replies, it needs to periodically check the log of decisions until the next checkpoint to see in which position on the finalized request log its request appears. It is similar to what blockchain clients do, inspecting the blockchain until their requests are included in a block a certain number of blocks before the blockchain head. This approach ensures clients would benefit from the extremely low latency of FLASHCONSENSUS as long as there are no more than $t_{fast}$ faulty replicas in the system.

### 4.3.3 Performance Degradation

To ensure FLASHCONSENSUS does not lead to performance degradation when compared to the performance of the conservative mode, all replicas periodically monitor their observed performance and compare it with expectations they have on the conservative mode. Replicas retrieve their expectations from AWARE's underlying latency prediction model [9]. Using this model, replicas can predict their consensus latency for the conservative mode using the network latency map and set their *consensus latency expectation threshold*. If they find that the consensus latency they currently observe exceeds this threshold, they stop their execution and ask for a synchronization phase. When $t + 1$ replicas ask for a synchronization phase, the system switches to the conservative mode (see next section). In the end, FLASHCONSENSUS only runs in fast mode if replicas observe the consensus latency to be lower than the expected latency in the conservative mode.

### 4.3.4 Reconfiguration of the System

As explained before, FLASHCONSENSUS operates in two regimes: *fast*, in which smaller quorums are used and $t_{fast}$ failures are tolerated, and *conservative*, in which standard-size quorums are employed and $t$ failures are tolerated.

The system starts in the conservative mode, and after finishing a predefined number of $u$ consensus instances successfully, it switches to the fast mode. Such reconfiguration is very simple because it is done deterministically at a certain point in the execution, i.e., after the consensus instance $i + u$ is decided. At this point, it simply requires each replica locally changing the fault threshold to $t_{fast}$ and recalculating its quorums size before executing the consensus instance $i + u + 1$.

A replica stays in the fast configuration until the underlying consensus' synchronization phase [54] (equivalent to PBFT's view change [17]) is triggered. 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 synchronization phase, which always runs considering threshold $t$, not $t_{fast}$ (which might already be violated).

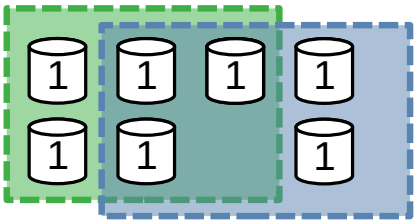
In the case of safety issues, when a new leader is elected, the first transaction of its regency after repairing the system to a single transaction history is a reconfiguration request [11] asking for the removal of the Byzantine replicas that participated in the detected equivocation. This request contains the PoC generated by the execution of the forensics protocol, as described in Section 4.3.1. Every correct replica removes these compromised replicas from the system after processing reconfiguration requests containing valid PoCs for such replicas.

Removing faulty replicas means the total number of replicas will now be smaller. The system needs to reconfigure $t$ and $t_{fast}$ accordingly. These two values can be equal in cases where the total number of replicas is not enough after the reconfiguration to have an optimistic case. For example, having $n = 7$, $t = 2$, $t_{fast} = 1$ and $\Delta = 0$, after detecting faulty behaviour in optimistic mode and removing $t_{fast} + 1$ faulty replicas, the reconfigured system will be left with $n = 5$, $t = 1$ and $\Delta = 1$. Since $t_{fast}$ must be at most $\frac{t}{2}$ and, in this example is impossible to have a number different than $t$ that satisfies this requirement, it is impossible to have an optimistic case in this extreme example (a representation can be seen in Figure 4.3).

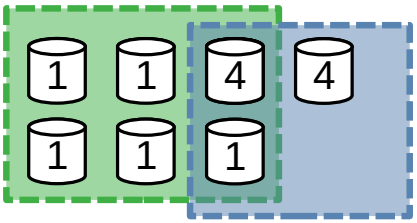
Further, in the case of equivocations, some replicas might need to roll back their states to a previous *stable checkpoint* (again, as discussed in Section 4.3.1), reapplying the correct transaction history as defined by the new leader on this state.

## 4.4 Ensuring Linearizability in FLASHCONSENSUS

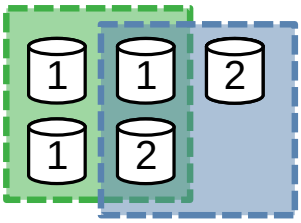
In typical BFT SMR systems, a client waits for $t + 1$ matching replies to ensure the replicated system perfectly emulates a centralized server, i.e., it satisfies linearizability [33]. This quorum size becomes $\lceil \frac{n+t+1}{2} \rceil$ if one wants to avoid running a consensus when performing read-only operations [8, 17]. These quorum sizes are still valid in FLASHCONSENSUS while in conserva-




(a) Conservative mode of operation quorums ($n = 7, t = 2, \Delta = 0$)




(b) Optimistic mode of operation quorums ($n = 7, t_{fast} = 1, \Delta = 3$)




(c) After reconfiguration quorums ($n = 5, t = 1, \Delta = 1$)

Figure 4.3: Weighted quorums reconfiguration example.

tive mode; however, when the system is in the fast mode, the existence of equivocations and the possibility of divergent decisions (that will be later detected and punished) requires revisiting the matching replies quorum expected by clients.

Figure 4.4 illustrates the situation where two clients received conflicting replies ($v$ and $v'$) from two different quorums ($Q$ and $Q'$, respectively) for some instance $i$ (ignore the leader change quorum for now). Even if $t$ malicious replicas are present in the intersection of the quorums, a client can assume that its request has been committed and will not be rolled back by waiting for $n - t$ matching replies. Due to the $n > 3t$ assumption, $(n - t) + (n - t) > n + t$. It means that two quorums with $n - t$ elements intersect in more than $t$ replicas and thus the intersection must contain not only faulty replicas. By waiting for $n - t$ matching replies, responses accepted by clients will never be rolled back, even with divergent decisions for consensus instance $i$, *as long as there are no leader changes*.

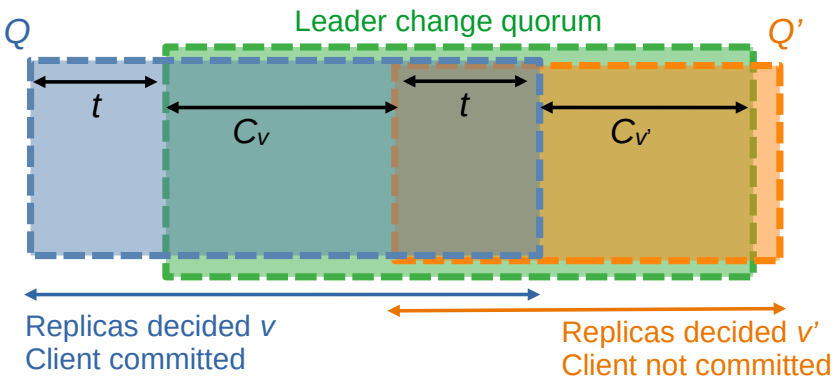



Figure 4.4: Quorum reasoning in FLASHCONSENSUS.

Now, consider the same scenario in which the replies for $v$ were deemed final by the client, but there was a leader change, and the new leader needs to define the result of consensus instance $i$. In this scenario, the elected leader waits for $n - t$ replicas to inform their status as indicated in the quorum of Figure 4.4, receiving replies from every replica but $t$ slow replicas that decided $v$.

In this setting, the decision for $v$ will be preserved as long as such value is the majority value among the ones informed by replicas, i.e., $C_v > C'_v + t$ in the figure. Considering $n = 2t + C_v + C'_v$ and $Q = C_v + 2t$ (both directly from the figure), we can reach that $C_v > n/3$, leading to $Q = n$. Therefore, waiting for $n - t$ matching replies is insufficient to ensure a value is not rolled back during a leader change when the system neutralizes the equivocation and turns back to the conservative mode. The only quorum big enough to ensure that is waiting for matching replies from all replicas.

Fortunately, by integrating BFT forensics [51] in the protocol, and in particular in the leader change sub-protocol, we can make such a quorum smaller. More specifically, we observe that to produce equivocations that lead some correct replicas to decide $v$ and $v'$, and later force a committed value to be rolled back, the $t$ equivocators must participate in the three quorums (for $v$, for $v'$ and leader change). Therefore, if the new leader executes the forensics protocol during the synchronization phase, it is possible to identify up to $e \leq t_{fast} + 1$ equivocators. Consequently, it is possible to discard the contributions of such malicious replicas and wait for messages from $e$ additional replicas. This situation is illustrated in Figure 4.5.

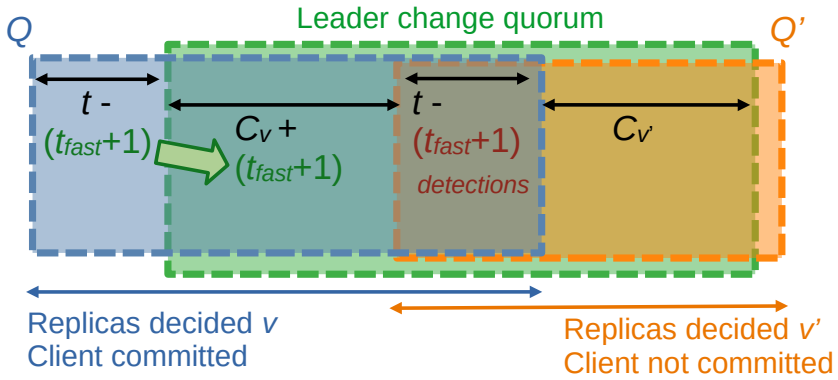



Figure 4.5: Waiting for information from $(t_{fast} + 1)$ more replicas during a leader change in FLASHCONSENSUS.

In this scenario, instead of assuming $C_v > C'_v + t$, we have $C_v + (t_{fast} + 1) > C'_v + t - (t_{fast} + 1)$. By developing this equation like before, we can reach a waiting quorum size of $n - t_{fast} - 1$, which defines the number of matching replies required for a client to know the result of its operation is durably committed, ensuring the linearizability of the replicated service.

## 4.5 Improving Latency with Speculation

As we just saw, preserving linearizability requires clients to collect $n - t_{fast} - 1$ replies, which is more than what is typically required in BFT SMR. This result is expected to negatively impact the

latency observed by clients in the fast mode.

FLASHCONSENSUS can further lower clients' observed latencies using the idea of client-side speculation. For this purpose, we implemented *correctables* [31] in the client shim of our BFT protocol. A correctable is a programming abstraction that allows a client application to work with incremental consistency guarantees and thus accelerates the application by allowing it to speculate with intermediate results. The state of a correctable can be updated multiple times, depending on the replies received by the replicas, strengthening the consistency guarantee each time, until it reaches the *final* state, which corresponds to the strongest consistency guarantee.

We design the FLASHCONSENSUS correctable following two principles: (1) to ensure the same safety guarantee as other BFT SMR protocols, the final consistency guarantee should satisfy linearizability under the maximum resilience threshold $t$, and (2) less safe consistency guarantees may relax either the assumptions on the number of Byzantine replicas or trade linearizability for a weaker consistency guarantee.

We define incremental consistency levels for FLASHCONSENSUS as follows (see Figure 4.6). *First* is the speculative result a client can access as soon as the first response from a replica arrives and does not provide any correctness guarantee. *Weak* demands responses from replicas totaling $Q_v = t_{fast} \cdot V_{max} + 1$ votes, and thus must have been confirmed by at least one correct replica if $f \leq t_{fast}$. The result can be stale, leading to the satisfaction of only sequential consistency under $t_{fast}$ failures. *Strong* demands $Q_v = 2t_{fast} \cdot V_{max} + 1$ weights and satisfies linearizability if $f \leq t_{fast}$. Since the read-only optimization requires such a larger quorum, a correctable in a strong state ensures linearizability as long as $f \leq t_{fast}$. Finally, the *Final* level satisfies linearizability under $t$ (just like any typical SMR with the read-only optimization enabled) by waiting for $n - t_{fast} - 1$ replies, as explained in the previous section. Since BFT SMR requires linearizability, only *Strong* and *Final* give the typical safety guarantee for their respective resilience thresholds.

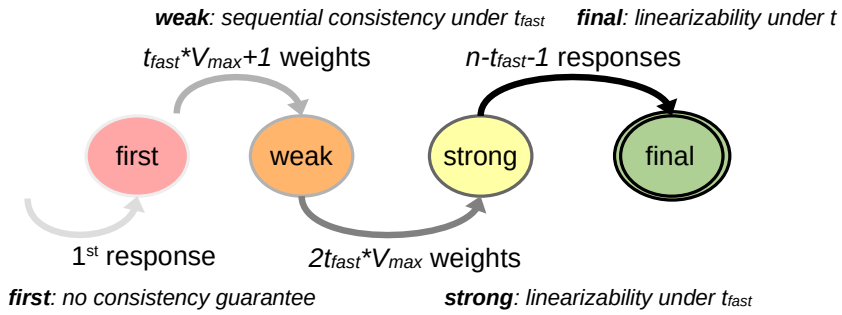



Figure 4.6: Incremental consistency levels that can be accessed through the correctable programming interface.

## 4.6 Final Remarks

Throughout this chapter, we covered the key aspects of our proposed solution. FLASHCONSENSUS is a self-optimized protocol that adapts the resilience threshold and replica weights to create

smaller quorums for low-latency transaction execution. We took advantage of abortable state machines to switch between a conservative and an optimistic mode of operation, together with a robust mechanism for diagnosing the system when there is more than $f$ faults.

To ensure linearizability as in standard SMR, we also needed to derive the exact number of matching replies a client needs to expect to preserve linearizability ($n - t_{fast} - 1$) even when consensus agreement violations are possible.

Additionally, we redefined the client-replica contracts for clients to tune the system for low latency or consistency under failures using *correctables*. FLASHCONSENSUS correctable has a set of incremental consistency levels (*First*, *Weak*, *Strong* and *Final*) that ensure the same safety guarantees as other BFT SMR protocols when using the *Final* consistency, and with the less safe consistency levels can relax either the assumptions on the number of Byzantine replicas tolerated or trade linearizability for weaker consistency guarantees.

In the next chapter, we will discuss the development of a prototype to test and assess our possible latency improvements.

# Chapter 5

# Implementation

We implemented a prototype of FLASHCONSENSUS on top of AWARE, an extension of BFT-SMART that BFT supports dynamic assignment of weights and leader placement. This implementation uses TLS to secure all communication channels and the elliptic curve digital signature algorithm (ECDSA) and SHA256 for signatures and hashes, respectively.

BFT-SMART and AWARE are libraries implemented using the Java programing language. The BFT-SMART project has 140+ classes implemented, adding up to 15000+ lines of code. AWARE extended this project, increasing the classes number to 150+ and lines of code to 17000+. FLASHCONSENSUS increased these numbers to 170+ classes and 19000+ lines of code.

Most of our modifications were related to the switching between two modes of operations (with different resilience thresholds) and implementing BFT forensics, which requires the signing of WRITE and REPLY messages (see Figure 2.2).

The implementation of FLASHCONSENSUS was divided into three main phases: Forensics, Mode of Operation switch and Correctables.

## 5.1 Forensics

In BFT-SMART, slight modifications to the protocol are necessary in order to support forensics. The first modification is that all WRITE messages need to be signed by their respective senders. All ACCEPT messages are already signed in the regular version of the protocol. Signing also the WRITE messages makes it possible to blame faulty replicas both in the ACCEPT and WRITE phase.

The second modification is that all replicas must save their respective quorum aggregate proofs at the end of both the WRITE and ACCEPT phases. These aggregates will contain the value and signatures received from other replicas that participated in the consensus and formed the quorum to advance to the next phase of the protocol. Using the intersection of these aggregates, we can blame faulty replicas (if they exist), as mentioned previously.

During the execution of BFT-SMART, a forensics round will take place periodically. During this round, one replica will request a set of quorum aggregate proofs saved by other replicas during the WRITE and ACCEPT phases and compare it with their own. By conducting these

rounds regularly, the protocol can identify any group of replicas that may attempt to manipulate the system. If faulty replicas are detected, the aggregate used for detection is dissiminated among the remaining replicas. A representation of one of these rounds can be seen in Figure 5.1.

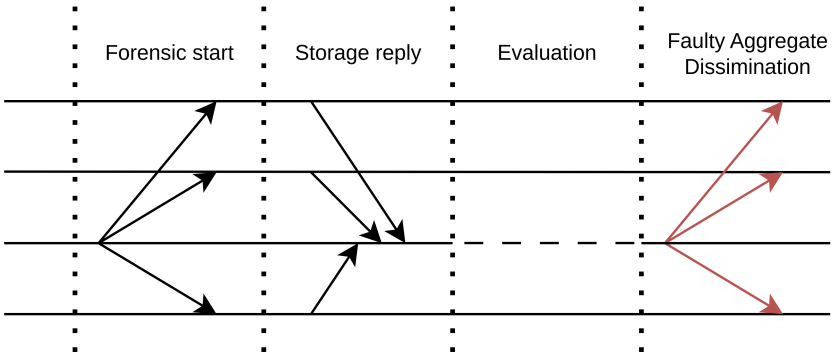



Figure 5.1: Forensics Protocol in BFT-SMART

To provide forensic support, we slightly modified an existing class already implemented in BFT-SMART source code.

**Acceptor.** This class represents the acceptor role in the consensus protocol. It is responsible for supplying an atomic multicast service. The Acceptor class can now receive and process messages related to the forensics support and signs all WRITE messages sent by the replica.

Together with the changes in the existing code, four new classes were implemented to allow forensics. In addition to these classes, two new messages were added to the protocol. These classes and messages descriptions can be seen in the following paragraphs.

**Aggregate.** The Aggregate class stores all proofs related to one consensus instance. For validation purposes, it needs to have the value decided in the consensus instance and all proofs received by replicas participating in the quorum. To store all the proofs, we used a Map with the sender id as key.

**Audit Storage.** The Audit Storage class is responsible for saving all aggregates created during the protocol. This class will have all WRITE certificates and ACCEPT certificates stored in two different Maps for easy access. The storage will be serialized and transmitted during the forensics execution and then compared with the storage of other replicas to detect any malicious behaviour.

**Auditor.** The Auditor class is responsible for receiving two different storages and comparing them. This class will be responsible for detecting faulty replicas if they exist.

**AuditorProvider.** The AuditorProvider class is an abstraction for the protocol. This class has access to the ServerViewController, one Auditor and the replica AuditStorage, and the last successfully executed audit (this will be explained further). This class is responsible for creating the

certificates and saving them in the replica AuditStorage. The Acceptor class only needs access to the AuditProvider and executes the corresponding method when an audit message is received.

Two new messages were also added to the protocol related to the forensics phase.

**AUDIT message.** The AUDIT message, with format $\langle \text{AUDIT}, cid, sid \rangle$, initiates the forensics protocol. These messages are relatively simple as they only contain the consensus and sender id ($cid$ and $sid$ respectively).

**STORAGE message.** The STORAGE message, with format $\langle \text{STORAGE}, cid, sid, St \rangle$, is the natural response for an AUDIT message. These messages contain the consensus and sender id, together with the sender AuditStorage ($St$). The AUDIT message sender will later process the received AuditStorage.

### 5.1.1 Multithreading

A replica should be able to respond to clients and participate in future consensus while performing forensics. For this effect, there needs to be some level of multithreading. In our implementation, the AuditProvider class is initialized with a pre-defined number of threads (AuditThreads) responsible for performing forensics. This number is defined in a configuration file. When an AuditStorage is received, it is placed in a BlockingQueue. This Queue needs to be a BlockingQueue to prevent concurrency problems. AuditThreads later access this Queue to process saved AuditStorage objects.

The number of threads is a crucial parameter for the system because the comparison of audit storages can be resource-intensive. The number of threads should be sufficient to complete a full forensics process before starting another one. However, it must not be too high, as this would divert resources from the overall consensus protocol. A thorough evaluation of this parameter and its impact on the overall protocol latency is presented in Section 6.2.

### 5.1.2 Garbage collection

Garbage collection is another crucial aspect of forensics. Throughout the consensus execution, there can be a massive number of messages that need to be stored for later audits. Keeping these messages can be very memory intensive. There needs to be a way to remove messages that are no longer necessary to prove the correctness of the system.

To remove unnecessary certificates from a replica storage, we wait for $2t + 1$ certificates. After evaluating $2t + 1$ certificates for a specific consensus instance and there was no malicious behaviour detected, it is safe to assume that the state in at least $2t + 1$ replicas is the same and certificates prior to that consensus id can be discarded since there is no need to prove correctness of the system. Doing so makes it possible to eventually remove older certificates from the system, preventing overloading the memory.

## 5.2 Mode of Operation Switch

Performing the switch between the two possible modes of operation requires several already implemented classes to be modified.

**ServerViewController.** This class is responsible for managing server-side information related to the current view and, if necessary, reconfiguring it to a new view. While operating in the conservative mode, this class was modified to compute the next possible faster view by reducing the threshold. Similarly, when operating in the optimistic mode, it was modified to calculate the next safest view by increasing the threshold.

**AWAREController.** This class was added to BFT-SMART when implementing AWARE and is responsible for the adaptive wide-area replication. It is the class responsible for computing the best weight configuration. After our modifications, if a threshold decrease is possible, it reconfigures the system to the best view with the threshold decrease using the ServerViewController class.

**DeliveryThread.** This class was modified to initiate an audit phase using the AwareController class when the storage with all the proofs reaches a specific size.

**Synchronizer.** This class implements the synchronization phase. The modifications performed in this class were regarding the switch between the optimistic and the conservative mode of operation. When a synchronization phase is initiated, indicating an issue has occurred, this class now switches to the conservative mode of operation (if not already in that mode) using the ServerViewController class.

### 5.2.1 Configuration

Our system only functions in two modes of operation, a conservative mode, tolerating the maximum faults possible, and an optimistic mode tolerating half the fault tolerated by the conservative mode. However, our implementation allows us to control how much the threshold decreases each time the system is optimized. It is possible to have more than two modes of operation with a conservative mode and several optimistic modes, each more optimistic than the previous one until the most optimistic mode tolerates half the maximum faults. This threshold reduction can be defined similarly to other parameters in a configuration file.

### 5.2.2 Locking mechanism

Throughout the testing of our prototype, an issue was identified. Sometimes after a reconfiguration to switch the mode of operation, the protocol would stop responding to clients. This problem appeared because replicas could not correctly halt the execution of consensus instances before reconfiguration happened since both reconfiguration and execution of consensus instances happen

in parallel. In order to fix this problem, similar to classes previously mentioned, another class was modified.

**TOMLayer.** This class implements the state machine replication protocol. One new ReentrantLock was added to this class to fix the locking problem. This lock controls when a reconfiguration is completed, allowing for the execution of consensus instances.

**AWAREController.** This class was changed further to use the new locking mechanism. When reconfiguration happens, the new TOMLayer ReentrantLock is used to lock and only unlock when the current reconfiguration is completed.

## 5.3 Correctable

Correctables were implemented in a reasonably simple way. There are two enumerations and one main class.

**CorrectableState.** Is an enumerate that represents the three states that a Correctable can assume (UPDATING, ERROR and FINAL).

**Consistency.** Is an enumerate that represents the different levels of consistency provided by our application. For an in-depth explanation, refer to Section 4.5.

**Correctable.** The Correctable class is responsible for delivering the result of one request invocation to our protocol by a client, considering the consistency level chosen. This class has the following attributes:

- CorrectableState - The current state of the Correctable. This state should change with the number of received messages;
- ClientViewController - This attribute contains both the current view $n$ and $t$. This information is essential since it is needed to calculate the appropriate number of votes required for responding to the client.
- votes - this represents the current amounts of votes. This value is appropriately incremented every time a replica receives a response taking into consideration the actual weight of the responding replica.
- responses - Similarly to the number of votes, this number is incremented every time a response is received from a replica.
- Semaphore - One semaphore used to wait in a blocking way for the desired number of replies.

### 5.3.1 Blocking behaviour

Continuously checking the state of Correctables can impact performance, as it would burden the client, who would have to continuously verify if the Correctable has received the desired number of votes to deliver a response. To mitigate this issue, we utilize a semaphore. Every time a response is received, the semaphore is signalled. When the desired number of responses is received, the semaphore is released. The Correctable then evaluates if the number of votes is sufficient to respond to the client and, if so, responds. If the Correctable does not have the needed votes, it will wait for another message replica response and then re-evaluate the number of votes.

However, this semaphore must be released after a timeout of waiting to prevent a deadlock. If there is a view change during the semaphore wait, and for example, some replicas are removed from the view, the Correctable will then wait for several replicas that do not exist. The Correctable needs to recalculate and scrap its current votes and responses received to reflect the changes from the view change. Using a timeout during the wait, if the said view change happens, it is possible to recalculate all new values, wait for the correct number of votes, and respond to the client accordingly.

### 5.3.2 Interface

A developer who intends to use our implementation of Correctables should be able to use the created interface in their desired client without any further modifications required. The details of the interface and its explanation are provided in the rest of this section for reference.

The Correctable constructor. This constructor requires the $ClientViewController$ since it contains crucial information on the view.

```
public Correctable(ClientViewController controller);
```

The following methods wait for a set quantity of votes and respond to the client with the protocol-decided value. The desirable Consistency defines the number of votes needed.

```
public byte[] getValueFirstConsistency();
public byte[] getValueWeakConsistency();
public byte[] getValueStrongConsistency();
public byte[] getValueFinalConsistency();
```

The client can also wait for a specified number of votes and responses not covered by previously mentioned Consistencies. This can be useful for some application-specific consistency requirements.

```
public byte[] getValue(double nVotes, int nResponses);
```

At any given time, the client can also ask the Correctable what is the current state (UPDATING, ERROR or FINAL).

```
public CorrectableState getState()
```

## 5.4 UML diagram

Figure 5.2 is a simplified UML Diagram with relevant classes mentioned in the previous sections.

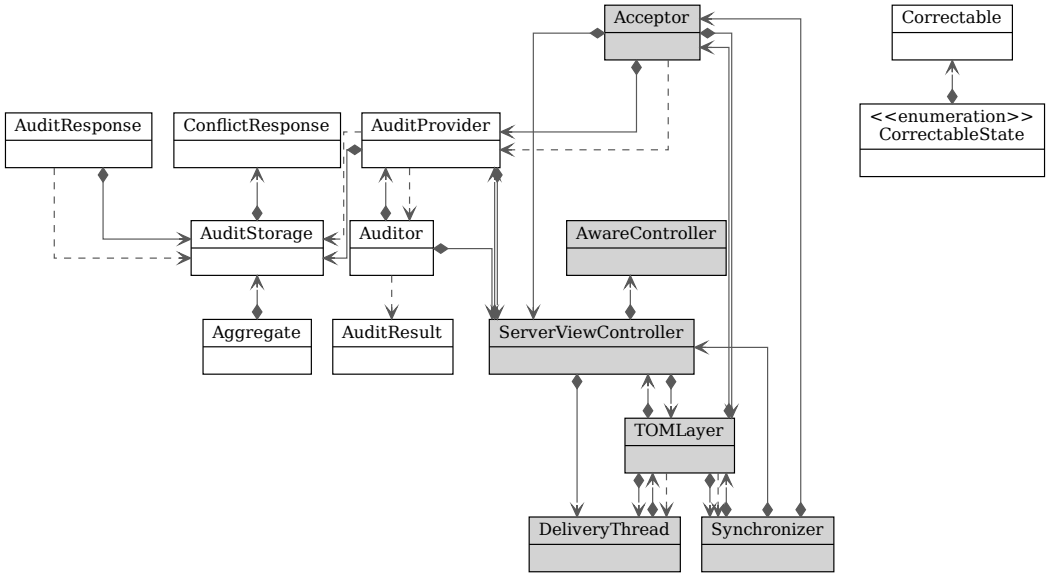



Figure 5.2: Simplified UML Diagram.

## 5.5 Final Remarks

This chapter detailed the implementation of FLASHCONSENSUS main components, showing all the additions and modifications made to previously developed libraries. We described the main components involved with forensics, the mode switch and the correctables.

We also provided a simplified class diagram with the main actors in our implementation. In this diagram, classes already existing in the BFT-SMART and AWARE libraries that were modified are represented in grey.

The next chapter will present an experimental evaluation done to assess the performance improvements brought by this implementation and the possible costs of performing forensics.

# Chapter 6

# Evaluation

In this chapter, we address the evaluation of FLASHCONSENSUS. The main focus of this evaluation is to understand to what extent our approach affected the overall performance of previously discussed protocols, either in throughput or latency.

Firstly to better grasp the impact of performing forensics in a protocol such as BFT-SMART, we conducted some latency and throughput tests between BFT-SMART and a version with only the forensics implementation without any change regarding the different modes of operation. Lastly, we use the full implementation of our prototype to investigate the potential performance gains of FLASHCONSENSUS, comparing it to AWARE and BFT-SMART as baselines. Later, we reason about FLASHCONSENSUS's runtime behaviour (particularly its adaptiveness) in the presence of faults or unfavorable network conditions. To compare the latency of FLASHCONSENSUS with AWARE and BFT-SMART, we configured an emulated network of 21 replicas based on real data from the internet. This network was emulated in LASIGE data center using a high-fidelity, state-of-the-art tool (Kollaps [30]) that can accurately emulate specific networks and provide results that can be used to predict real-world behaviour. Our experiments focus mostly on measuring latency, which is fundamentally limited by the quality of the links and quorum formation rules in a wide-area network.

## 6.1 Evaluation Setup

All tests performed in this chapter were executed in 16 machines with the following characteristics: Dell PowerEdge R410 with two quad-core Intel Xeon E5520 (2.27 GHz) CPUs; two hardware threads per core; 32 GB of RAM; Ubuntu 20.04 operating system with OpenJDK RE 17.0.3. The machines are connected through a gigabit ethernet.

We emulate a wide-area network with network characteristics that resemble the AWS cloud infrastructure. For this purpose, we employ the Kollaps network emulator, which was validated for realistic WAN experimentation with BFT-SMART and WHEAT in [30], and use round-trip latency statistics from cloudping[1] to construct the emulated network (for detailed information on specific values, please refer to Tables A.1,A.2,A.3 and A.4).

[1] `https://www.cloudping.co/grid`.

In our setup, unless noted otherwise, we deploy a client and a replica in each of the $n = 21$ AWS regions (depicted in Figure 1.1a). All clients send 400-bytes requests simultaneously and continuously to the replicas (2000 per client) until each has finished its measurements. A client request arriving at the leader may wait until it gets included in a batch when there is currently a consensus instance running. We employ synchronous clients that block until a result is obtained and send the next request after randomly waiting for up to $1s$. Finally, *request latency* is the average end-to-end protocol latency computed by a client after completing all operations.

## 6.2 Forensics Impact

We conducted both latency and throughput tests to evaluate the impact of performing forensics.

In these experiments, we executed forensics for every 500 consensus instances and allocated eight threads responsible for comparing the different audit storages received.

**Throughput.** In throughput testing, we used from 21 to 3150 clients evenly distributed throughout all available regions on our emulated network. For each round, all clients continuously send requests to overload the system. We also experimented with requests with a total size of 400 bytes and zero bytes. The results can be seen in Figure 6.1a.

The difference between running the protocol with or without forensics does not significantly impact throughput, decreasing the throughput by 0.04%. As expected, requests with zero bytes yield slightly higher overall throughput, increasing throughput by 7%.

**Latency.** In latency testing, we deployed one client located in the same region as our current leader. The results can be seen in Figure 6.1b.

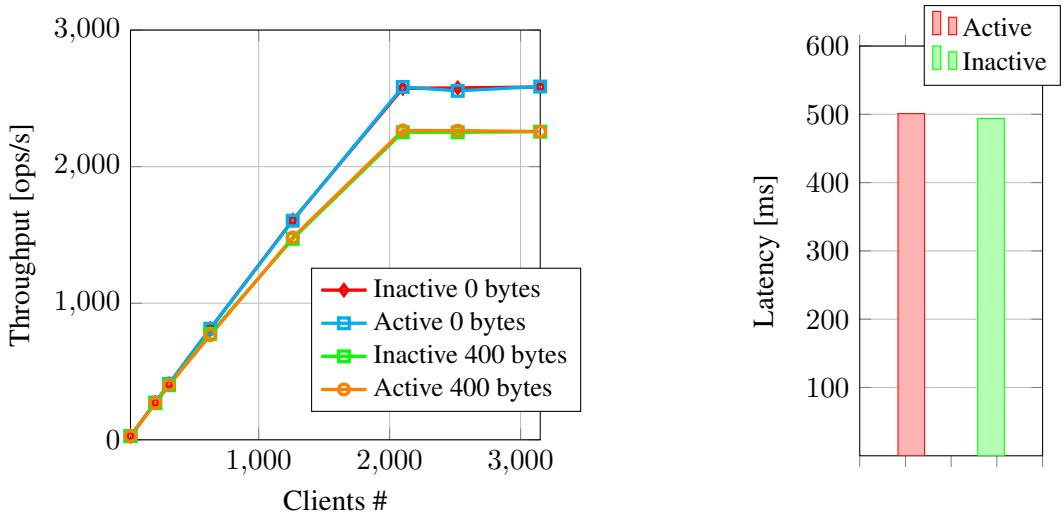
Not using forensics yields slightly better results in terms of latency. This is to be expected since all machines are using more resources. However, the difference in latency is marginal (less than 1%).

**Resources usage.** In the next test, we compare the use of CPU and memory with and without the use of forensics. The results can be seen in Figure 6.2.

As expected, the use of resources increases when using forensics in the BFT-SMaRt protocol. There is an increase of 39% in CPU usage and 8% in memory usage. The higher usage of CPU resources is primarily due to the increased number of cryptographic operations required. The additional memory usage is required to store all of the proofs acquired throughout the protocol.

**Multithreading impact.** In this test, we aim to find the minimum possible number of threads allocated to audit that do not jeopardize the protocol's normal functioning while ensuring that an audit phase can be completed before the next one begins.

In our experiment, we used one, four and eight threads responsible for audit and evaluated two different aspects:




(a) Throughput comparison.

(b) Latency comparison.

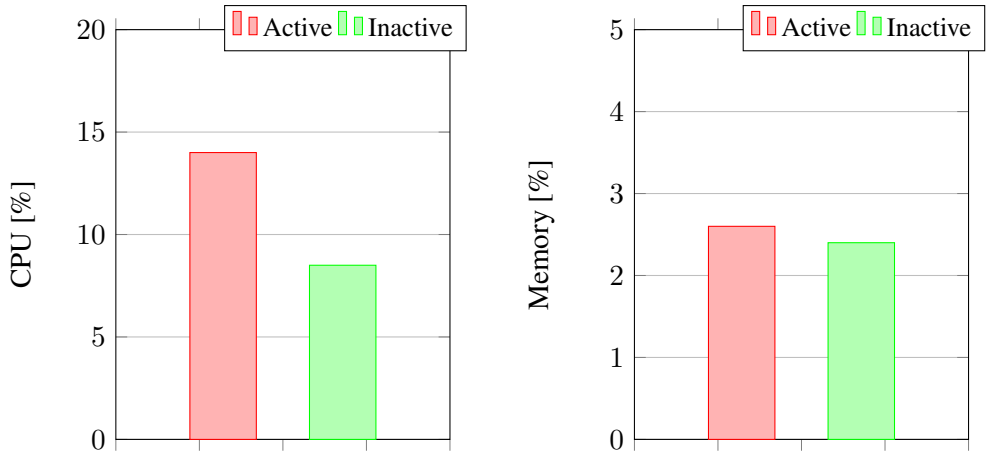
Figure 6.1: How forensics affect the performance of BFT-SMaRt ($n = 21$).

1. The average time an audit round takes to complete, e.g. comparing $2t + 1$ received audit storages to ensure no malicious behaviour happened between the tested consensus. These results can be seen in Figure 6.3a. As expected, the higher the number of threads responsible for auditing, the faster it is possible to finish a round since several audit storages can be compared in parallel. The time required to finish a round decreases by 47% when going from one thread to four, and by 8% when going from four threads to eight.

2. The behaviour witnessed by one of the clients throughout the execution of the protocol with a different number of threads. These results can be seen in Figure 6.3b. Our results conclude that in our specific testing environment, one or four threads are insufficient in a system with continuous requests. After several requests, the application can not keep up with requests because it is still trying to finish comparing audit storages. Because of the delay and the high number of audit storages on hold for comparison, the application eventually overloads and stops responding to clients. We can see both experiments with one thread and four threads eventually stop responding, outputting a latency of 5000 ms (timeout for client requests). Using eight threads, even though there is a slight increase in latency (with is expected since forensics rounds are being performed in parallel with the protocol), the application can complete audit rounds before the subsequent rounds begin, and the application keeps functioning normally.

## 6.3 FlashConsensus Acceleration

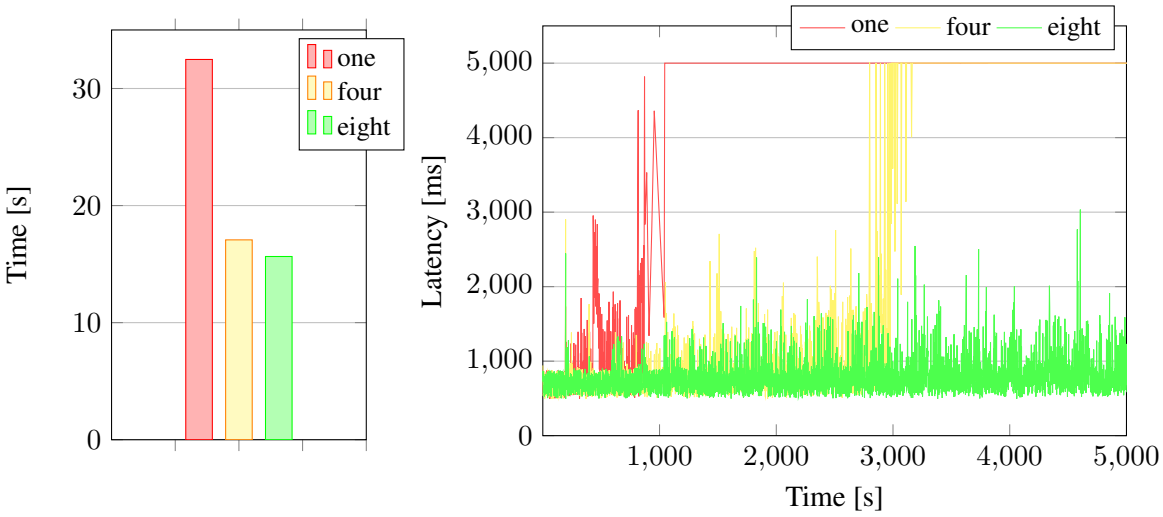
We evaluate FlashConsensus, AWARE, and BFT-SMaRt in our emulated AWS network. For a better exposition, we group the 21 clients' results by the continent they are located in, reporting only their regional averages (see Figure 6.4). We observe that FlashConsensus significantly




(a) CPU comparison.

(b) Memory comparison.

Figure 6.2: How forensics affect CPU and memory usage




(a) Audit round average duration.

(b) Thread number behaviour comparison.

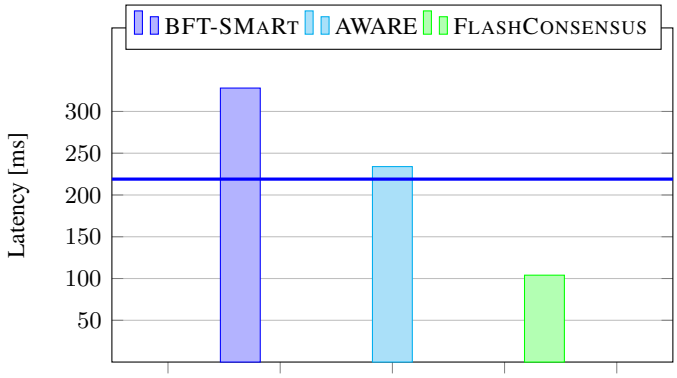
Figure 6.3: Thread number impact comparison.

accelerates consensus, leading to a speedup of $3.15\times$ faster decisions (see Figure 6.4a). This result also surpasses the speedup of $1.5\times$, achievable if the speed of the links employed by BFT-SMaRt approximated the speed of light.[2] Second, accelerating consensus decisions also leads to faster request latencies observed by clients worldwide (see Figure 6.4b). Averaged over all client regions, FlashConsensus leads to a speedup of $1.92\times$ over BFT-SMaRt (AWARE leads to $1.39\times$).

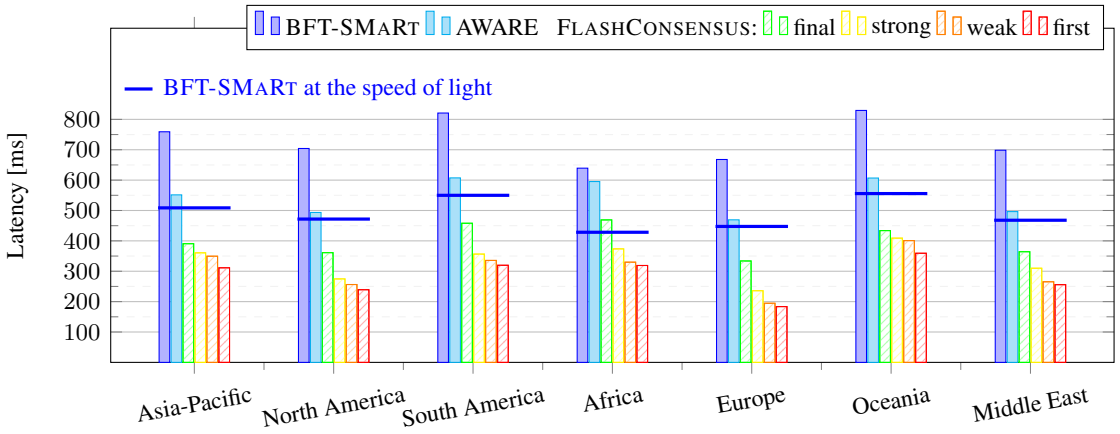
Our results also show that even higher speedups can be achieved by employing the incremental consistency levels of FlashConsensus's correctables. The *strong* consistency level, which

[2] Assuming all links transmit at the speed of $0.67c$ [14, 36].

guarantees linearizability if $f < t_{fast}$, achieves a speedup of $2.42\times$, while the speculative levels *weak* and *first* achieve speedups of $2.70\times$ and $2.91\times$, respectively (results are averaged over all regions).




(a) Consensus latency.




(b) Clients' observed end-to-end latencies for protocol runs with BFT-SMaRt, AWARE and FlashConsensus. The client results are averaged over all regions per continent.

Figure 6.4: Achievable latency gains for the $n = 21$ AWS setup.

## 6.4 FlashConsensus Throughput

Although FlashConsensus aims to optimize latency, we conducted a simple 0/0-microbenchmark with an increasing number of clients evenly distributed among all AWS regions while measuring the throughput of BFT-SMaRt, AWARE ($t = 6$), AWARE ($t = 3$), and FlashConsensus. In this experiment, we used 0-byte requests to avoid the saturation of links bandwidth. Figure 6.5 presents the results.

These results show two main insights. First, faster consensus instances can achieve higher throughput, provided the available network bandwidth is not exhausted. FlashConsensus,

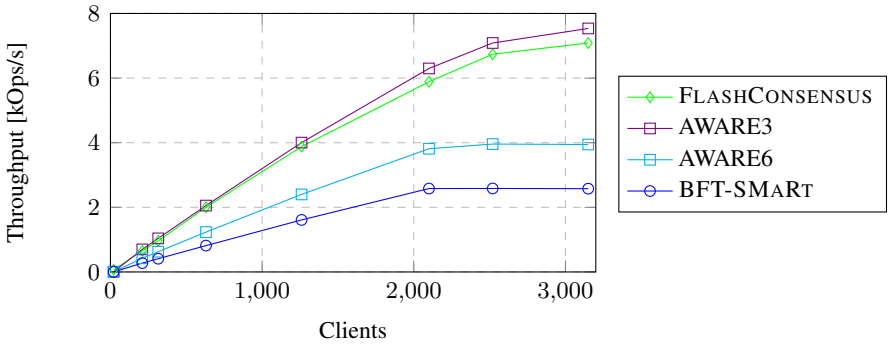



Figure 6.5: Throughput comparison for 0-byte requests and the $n = 21$ AWS setup.

compared to BFT-SMART and AWARE ($t = 6$), has a throughput increase of 59% and 38%, respectively. Second, FLASHCONSENSUS displays a minor penalty compared to AWARE ($t = 3$) (FLASHCONSENSUS throughput is 5% lower), which uses the same quorums but offers half of our resilience. This difference stems from the costs of additional signatures needed to integrate BFT forensics in our protocol.

## 6.5 FLASHCONSENSUS Runtime Behaviour

We launch FLASHCONSENSUS in the same $n = 21$ AWS setting to observe its runtime behaviour during the system's lifespan. Noticeably, clients' request latencies show high variations, which are caused by a random waiting time of a request at the leader before getting included in the next batch, which takes a varying time depending on how shortly the request arrived before the next consensus can be started. Moreover, we induce events to evaluate FLASHCONSENSUS's reactions (see Figure 6.6, which is plotted from a single representative correct client's perspective):

(1) Starts in the conservative configuration, displaying low performance.

(2) The system switches to the fast configuration (such switches are attempted every 400 consensus instances), leading to a significant latency improvement.

(3) The leader crashes.

(4) Replicas perform a leader change and abort, switching back to the conservative mode.

(5) After finishing 400 consensus instances again, replicas return to the fast mode. This leads only to a modest performance improvement because the weights have not been optimized yet.

(6) The system self-optimizes to its best weight distribution and leader placement (using the mechanisms inherited from AWARE), reaching again the low latencies experienced before.

(7) We artificially make the current leader slower by using the *tc* command. Increasing the response time from the leader to all other machines.

(8) FLASHCONSENSUS detects it is running in a sub-optimal configuration and changes replicas' weights and leader location to accelerate performance (the leader is changed from *eu-west-2* to *us-east-1*).

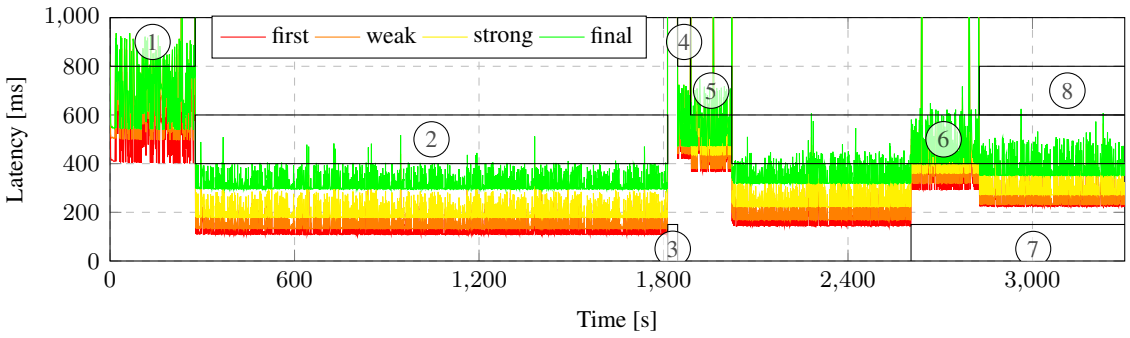



Figure 6.6: Runtime behaviour of FLASHCONSENSUS.

## 6.6 Final Remarks

In this chapter, we presented the performance evaluation of many aspects of FLASHCONSENSUS.

We first evaluated the overall impact of executing forensics in our system in terms of CPU and memory used. The results were as expected. Forensics puts a slight strain on both the CPU and memory. Secondly, we evaluated both latency and throughput. The results were positive, with FLASHCONSENSUS improving latency to up to half that of BFT-SMART. Lastly, we evaluated the runtime behaviour of FLASHCONSENSUS with leader crashes and network changes. Our protocol behaved as expected.

Using all these experiments, we can conclude that our protocol improved the performance of previously mentioned protocols.

# Chapter 7

# Conclusion

In this work, we presented the design and implementation of FLASHCONSENSUS, which aims to accelerate planetary-scale Byzantine consensus. FLASHCONSENSUS achieves this acceleration by combining weighted replication with BFT forensics, enabling the system to underestimate the resilience threshold safely. By utilizing faster quorums, FLASHCONSENSUS drives consensus decisions while ensuring the safety and integrity of the overall process. Additionally, we conducted a comprehensive evaluation to assess the potential performance improvements offered by FLASHCONSENSUS.

We showed how to obtain FLASHCONSENSUS from AWARE, by incorporating the ideas of abortable state machine replication, BFT forensics, and incremental consistency guarantees. Notably, FLASHCONSENSUS always achieves linearizability and liveness under the optimal resilience threshold – even when agreement quorums are formed using the fast threshold.

Our evaluation results indicate that latency benefits are substantial, i.e., achieving a speedup of $1.92\times$ over BFT-SMART. We also see potential at the client side speculation, which allows the application to choose a relaxed consistency level from the client-replica contract on the granularity level of individual operations. This can be a good fit for operations that are time-sensitive and not security-critical as they can benefit from latency speedups of up to $6\times$.

This thesis uses forensics in the protocol's optimistic fast mode to detect existing culpability and then remove and reconfigure the system accordingly. This culpability is detected after the system has been compromised, and a rollback is needed. A possible future work could be to find ways to integrate forensics in a more seamless way to function at the same time as the consensus, detecting malicious behaviour as it is happening without the need to roll back after the optimistic mode is compromised. Rather than relying on forensics as a deterrent to malicious attackers, it could serve as a proactive measure to prevent such attacks from occurring in the first place. However, this application should not be trivial since it needs to be carefully integrated into the protocol to not significantly harm the obtained performance gains.

As future work, we also want to integrate FLASHCONSENSUS in a blockchain to assess the possible advantages of using this protocol and compare it with existing blockchains.

# Bibliography


[1] Salem Alqahtani and Murat Demirbas. BigBFT: A multileader Byzantine fault tolerance protocol for high throughput. In *Proc. of the IEEE Int. Performance, Computing, and Communications Conference (IPCCC)*, pages 1–10, 2021.

[2] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling Byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 7(1):80–93, 2010.

[3] Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. Leaderless consensus. In *Proc. of the 41st IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 392–402, 2021.

[4] Balaji Arun and Binoy Ravindran. DuoBFT: Resilience vs. efficiency trade-off in Byzantine fault tolerance. *preprint arXiv:2010.01387*, 2020.

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

[6] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. RBFT: Redundant Byzantine fault tolerance. In *Proc. of the 33rd IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 297–306, 2013.

[7] Jean-Paul Bahsoun, Rachid Guerraoui, and Ali Shoker. Making BFT protocols really adaptive. In *Proc. of the 29th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, pages 904–913, 2015.

[8] Christian Berger, Hans P Reiser, and Alysson Bessani. Making reads in BFT state machine replication fast, linearizable, and live. In *Proc. of the 40th IEEE Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 1–12, 2021.

[9] Christian Berger, Hans P. Reiser, João Sousa, and Alysson Neves Bessani. AWARE: Adaptive wide-area replication for fast and resilient Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2020.

[10] Christian Berger, Lívio Rodrigues, Hans Reiser, Vinicius Cogo, and Alysson Bessani. Beating the speed of light: Low-latency planetary-scale adaptive Byzantine consensus. *ConsensusDay*, 2023.

[11] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proc. of the 44th Annu. IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, pages 355–362, 2014.

[12] Erica Blum, Jonathan Katz, and Julian Loss. Network-agnostic state machine replication. *preprint arXiv:2002.03437*, 2020.

[13] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. `https://ethereum.org/whitepaper/`, 2013. Accessed: August 3, 2023.

[14] Frank Cangialosi, Dave Levin, and Neil Spring. Ting: Measuring and exploiting latencies between all Tor nodes. In *Proc. of the ACM Internet Measurement Conference (IMC)*, page 289–302, 2015.

[15] Carlos Carvalho, Daniel Porto, Luís Rodrigues, Manuel Bravo, and Alysson Bessani. Dynamic adaptation of Byzantine consensus protocols. In *Proc. of the 33rd Annual ACM Symposium on Applied Computing (SAC)*, pages 411–418, 2018.

[16] Daniel Cason, Enrique Fynn, Nenad Milosevic, Zarko Milosevic, Ethan Buchman, and Fernando Pedone. The design, architecture and performance of the Tendermint blockchain network. In *Proc. of the 40th IEEE Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 23–33, 2021.

[17] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, page 173–186, 1999.

[18] Tairi Chiba, Ren Ohmura, and Junya Nakamura. Network bandwidth variation-adapted state transfer for geo-replicated state machines and its application to dynamic replica replacement. *preprint arXiv:2204.08656*, 2022.

[19] Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable Byzantine agreement. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 403–413, 2021.

[20] Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, and Jovan Komatovic. As easy as ABC: Optimal (A)ccountable (B)yzantine (C)onsensus is easy! In *Proc. of the 37th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 560–570, 2022.

[21] Paulo Coelho and Fernando Pedone. Geographic state machine replication. In *Proc. of the 37th IEEE Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 221–230, 2018.

[22] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red Belly: A secure, fair and scalable open blockchain. In *Proc. of the 42nd IEEE Symp. on Security and Privacy (SP)*, pages 466–483, 2021.

[23] Tobias Distler, Christian Cachin, and Rüdiger Kapitza. Resource-efficient Byzantine fault tolerance. *IEEE Transactions on Computers (TC)*, 65(9):2807–2819, 2015.

[24] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[25] Michael Eischer and Tobias Distler. Latency-aware leader selection for geo-replicated Byzantine fault-tolerant systems. In *Proc. of the 1st Workshop on Byzantine Consensus and Resilient Blockchains (BCRB)*, pages 140–145, 2018.

[26] Michael Eischer and Tobias Distler. Resilient cloud-based replication with low latency. In *Proceedings of the 21st International Middleware Conference*, pages 14–28, 2020.

[27] Michael Eischer, Benedikt Straßner, and Tobias Distler. 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)*, pages 1–9. 2020.

[28] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. In *Proc. of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, pages 1—-7, 1983.

[29] R. Friedman, A. Mostefaoui, and M. Raynal. Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2(1):46–56, 2005.

[30] Paulo Gouveia, João Neves, Carlos Segarra, Luca Liechti, Shady Issa, Valerio Schiavoni, and Miguel Matos. Kollaps: decentralized and dynamic topology emulation. In *Proc. of the 15th European Conference on Computer Systems (EuroSys)*, pages 1–16, 2020.

[31] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In *Proc. of the 12th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 169–184, 2016.

[32] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable and decentralized trust infrastructure. In *Proc. of the 49th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, pages 568–580, 2019.

[33] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[34] Heidi Howard, Aleksey Charapko, and Richard Mortier. Fast flexible paxos: Relaxing quorum intersection for fast paxos. In *International Conference on Distributed Computing and Networking 2021*, pages 186–190, 2021.

[35] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[36] Katharina Kohls and Claudia Diaz. Verloc: Verifiable localization in decentralized systems. In *Proc. of the 31st USENIX Security Symposium (USENIX Security)*, pages 2637–2654, 2022.

[37] Petr Kuznetsov, Andrei Tonkikh, and Yan X Zhang. Revisiting optimal resilience of fast Byzantine consensus. In *Proc. of the 40th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 343–353, 2021.

[38] Leslie Lamport. Proving the correctness of multiprocess programs. 1977.

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

[40] Shengyun Liu and Marko Vukolić. Leader set selection for low-latency geo-replicated state machine. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(7):1933–1946, 2017.

[41] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.

[42] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, pages 369–384, 2008.

[43] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Towards low latency state machine replication for uncivil wide-area networks. In *Proc. of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[44] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 3(3):202–215, 2006.

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

[46] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable BFT consensus with pipelined tree-based dissemination and aggregation. In *Proc. of the 28th ACM SIGOPS Symp. on Operating Systems Principles (SOSP)*, pages 35–48, 2021.

[47] Ray Neiheiser, Luciana Rech, Manuel Bravo, Luís Rodrigues, and Miguel Correia. Fireplug: Efficient and robust geo-replication of graph databases. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 31(8):1942–1953, 2020.

[48] Martin Nischwitz, Marko Esche, and Florian Tschorsch. Raising the AWAREness of BFT protocols for soaring network delays. In *Proc. of the 47th IEEE Conf. on Local Computer Networks (LCN)*, pages 387–390, 2022.

[49] Shota Numakura, Junya Nakamura, and Ren Ohmura. Evaluation and ranking of replica deployments in geographic state machine replication. In *Proc. of the 38th IEEE Int. Symp. on Reliable Distributed Systems Workshops (SRDSW)*, pages 37–42, 2019.

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

[51] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. BFT protocol forensics. In *Proc. of the 28th ACM Conference on Computer and Communications Security (CCS)*, pages 1722–1743, 2021.

[52] Douglas Simoes Silva, Rafal Graczyk, Jérémie Decouchant, Marcus Völp, and Paulo Esteves-Verissimo. Threat adaptive Byzantine fault tolerant state-machine replication. In *Proc. of the 40th IEEE Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 78–87, 2021.

[53] Yee Jiun Song and Robbert van Renesse. Bosco: One-step Byzantine asynchronous consensus. In *Proc. of the 22nd International Symposium on Distributed Computing (DISC)*, pages 438–450, 2008.

[54] João Sousa and Alysson Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proc. of the 9th IEEE European Dependable Computing Conf. (EDCC)*, pages 37–48, 2012.

[55] João Sousa and Alysson Bessani. Separating the wheat from the chaff: An empirical design for geo-replicated state machines. In *Proc. of the 34th IEEE Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 146–155, 2015.

[56] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolić. Mir-BFT: High-throughput BFT for blockchains. *preprint arXiv:1906.05552*, 2019.

[57] Xiao Sui, Sisi Duan, and Haibin Zhang. Marlin: Two-phase BFT with linearity. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 54–66, 2022.

[58] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. EBAWA: Efficient Byzantine agreement for wide-area networks. In *Proc. of the 12th IEEE Int. Symp. on High Assurance Syst. Eng. (HASE)*, pages 10–19, 2010.

[59] Benjamin Wester, James A Cowling, Edmund B Nightingale, Peter M Chen, Jason Flinn, and Barbara Liskov. Tolerating latency in replicated state machines through client speculation. In

*Proc. of the 6th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 245–260, 2009.

[60] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proc. of the 38th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 347–356, 2019.

# Appendix A

# AWS Latency Matrix

| ID | af-south-1 | ap-east-1 | ap-northeast-1 | ap-northeast-2 | ap-northeast-3 |
|---|---|---|---|---|---|
| af-south-1 | 9.09 | 353.99 | 354.69 | 382.53 | 359.96 |
| ap-east-1 | 356.59 | 1.47 | 50.35 | 36.91 | 39.86 |
| ap-northeast-1 | 359.51 | 51.45 | 4.27 | 34.26 | 11.32 |
| ap-northeast-2 | 385.17 | 37.81 | 35.28 | 2.83 | 30.09 |
| ap-northeast-3 | 362.68 | 39.28 | 10.04 | 33.01 | 4.71 |
| ap-south-1 | 277.31 | 101.57 | 126.73 | 131.52 | 119.3 |
| ap-southeast-1 | 317.25 | 42.2 | 72.94 | 77.48 | 68.88 |
| ap-southeast-2 | 414.5 | 132.99 | 108.45 | 147.91 | 115.65 |
| ca-central-1 | 228.91 | 190.73 | 143.62 | 172.46 | 149.34 |
| eu-central-1 | 162.4 | 195.74 | 222.35 | 231.06 | 224.17 |
| eu-north-1 | 182.75 | 215.79 | 242.9 | 248.62 | 244.6 |
| eu-south-1 | 172.45 | 190.69 | 214.58 | 220.13 | 217.41 |
| eu-west-1 | 165.0 | 221.94 | 201.67 | 230.0 | 207.29 |
| eu-west-2 | 154.38 | 210.7 | 210.19 | 240.94 | 215.84 |
| eu-west-3 | 152.28 | 203.95 | 216.62 | 244.48 | 222.53 |
| me-south-1 | 256.43 | 123.55 | 159.68 | 163.26 | 151.9 |
| as-east-1 | 347.78 | 305.98 | 258.54 | 286.89 | 264.53 |
| us-east-1 | 231.04 | 192.58 | 145.41 | 174.6 | 153.41 |
| us-east-2 | 241.24 | 179.58 | 132.29 | 164.06 | 138.17 |
| us-west-1 | 293.97 | 149.94 | 108.23 | 137.15 | 109.46 |
| us-west-2 | 275.37 | 138.74 | 98.22 | 127.51 | 99.01 |

Table A.1: AWS matrix - part 1.

| ID | ap-south-1 | ap-southeast-1 | ap-southeast-2 | ca-central-1 | eu-central-1 |
|---|---|---|---|---|---|
| af-south-1 | 272.83 | 309.73 | 411.21 | 224.61 | 155.92 |
| ap-east-1 | 93.92 | 44.26 | 134.0 | 192.22 | 197.24 |
| ap-northeast-1 | 129.34 | 71.18 | 106.39 | 144.9 | 223.12 |
| ap-northeast-2 | 127.75 | 74.51 | 144.92 | 171.59 | 231.01 |
| ap-northeast-3 | 119.0 | 69.06 | 115.77 | 149.53 | 226.9 |
| ap-south-1 | 1.84 | 56.55 | 148.36 | 188.03 | 117.18 |
| ap-southeast-1 | 58.37 | 3.63 | 93.59 | 210.42 | 156.98 |
| ap-southeast-2 | 151.53 | 95.23 | 3.14 | 199.37 | 248.18 |
| ca-central-1 | 188.32 | 210.25 | 198.63 | 3.71 | 91.76 |
| eu-central-1 | 120.06 | 158.45 | 250.04 | 92.75 | 3.48 |
| eu-north-1 | 138.72 | 177.61 | 295.19 | 107.57 | 23.25 |
| eu-south-1 | 108.88 | 149.23 | 239.24 | 103.91 | 12.71 |
| eu-west-1 | 123.08 | 181.35 | 258.56 | 71.15 | 25.43 |
| eu-west-2 | 114.42 | 172.46 | 268.14 | 78.24 | 17.18 |
| eu-west-3 | 106.04 | 165.42 | 279.2 | 86.16 | 13.0 |
| me-south-1 | 38.85 | 85.25 | 186.7 | 161.69 | 90.16 |
| as-east-1 | 302.99 | 323.74 | 313.46 | 126.42 | 208.69 |
| us-east-1 | 188.73 | 211.27 | 198.52 | 17.63 | 92.01 |
| us-east-2 | 197.03 | 198.64 | 187.99 | 25.04 | 101.7 |
| us-west-1 | 226.72 | 176.02 | 139.69 | 79.81 | 152.39 |
| us-west-2 | 215.91 | 166.68 | 140.36 | 60.31 | 142.86 |

Table A.2: AWS matrix - part 2.

| ID | eu-north-1 | eu-south-1 | eu-west-1 | eu-west-2 | eu-west-3 | me-south-1 |
|---|---|---|---|---|---|---|
| af-south-1 | 179.74 | 168.53 | 159.13 | 150.66 | 147.05 | 274.47 |
| ap-east-1 | 217.67 | 189.36 | 221.66 | 209.27 | 207.53 | 129.3 |
| ap-northeast-1 | 244.35 | 215.45 | 201.54 | 212.36 | 216.51 | 161.79 |
| ap-northeast-2 | 250.26 | 220.0 | 230.93 | 239.3 | 247.82 | 167.26 |
| ap-northeast-3 | 245.75 | 215.28 | 206.42 | 218.74 | 222.07 | 151.81 |
| ap-south-1 | 138.43 | 107.78 | 122.32 | 115.03 | 106.77 | 39.23 |
| ap-southeast-1 | 177.69 | 150.41 | 180.55 | 171.79 | 170.11 | 84.13 |
| ap-southeast-2 | 296.82 | 240.14 | 257.72 | 265.19 | 279.89 | 184.14 |
| ca-central-1 | 109.12 | 103.96 | 69.83 | 79.06 | 84.92 | 162.33 |
| eu-central-1 | 24.4 | 11.29 | 26.23 | 17.37 | 11.0 | 94.34 |
| eu-north-1 | 3.4 | 31.59 | 42.74 | 33.13 | 31.83 | 107.03 |
| eu-south-1 | 31.66 | 2.6 | 34.45 | 26.28 | 20.52 | 92.7 |
| eu-west-1 | 44.62 | 35.65 | 4.13 | 13.32 | 19.54 | 101.68 |
| eu-west-2 | 34.26 | 26.98 | 13.11 | 4.08 | 9.95 | 89.73 |
| eu-west-3 | 32.85 | 21.04 | 18.56 | 12.57 | 4.62 | 83.7 |
| me-south-1 | 107.47 | 94.19 | 96.32 | 87.12 | 81.3 | 2.58 |
| as-east-1 | 216.92 | 219.56 | 178.79 | 189.52 | 197.83 | 281.12 |
| us-east-1 | 115.18 | 101.49 | 67.16 | 78.4 | 82.46 | 162.15 |
| us-east-2 | 127.8 | 117.43 | 77.32 | 90.09 | 93.8 | 170.41 |
| us-west-1 | 175.08 | 161.55 | 138.08 | 147.1 | 143.11 | 228.87 |
| us-west-2 | 159.46 | 150.88 | 119.15 | 128.99 | 134.26 | 214.62 |

Table A.3: AWS matrix - part 3.

| ID | as-east-1 | us-east-1 | us-east-2 | us-west-1 | us-west-2 |
|---|---|---|---|---|---|
| af-south-1 | 344.3 | 227.63 | 237.44 | 288.13 | 274.42 |
| ap-east-1 | 304.4 | 193.02 | 179.63 | 146.87 | 137.88 |
| ap-northeast-1 | 257.43 | 146.19 | 132.08 | 110.24 | 98.53 |
| ap-northeast-2 | 286.84 | 174.39 | 161.06 | 138.04 | 127.76 |
| ap-northeast-3 | 262.84 | 151.93 | 137.64 | 112.31 | 99.23 |
| ap-south-1 | 302.68 | 187.92 | 197.17 | 225.9 | 217.35 |
| ap-southeast-1 | 323.19 | 211.58 | 200.1 | 175.55 | 164.61 |
| ap-southeast-2 | 313.81 | 199.08 | 187.38 | 140.42 | 140.21 |
| ca-central-1 | 127.44 | 16.62 | 25.05 | 79.44 | 61.85 |
| eu-central-1 | 208.17 | 92.14 | 101.66 | 151.67 | 143.81 |
| eu-north-1 | 216.65 | 111.53 | 121.38 | 174.95 | 159.55 |
| eu-south-1 | 218.18 | 102.02 | 111.01 | 161.06 | 151.3 |
| eu-west-1 | 178.56 | 67.3 | 76.68 | 137.72 | 118.59 |
| eu-west-2 | 187.51 | 77.28 | 86.56 | 148.29 | 128.72 |
| eu-west-3 | 200.86 | 84.0 | 92.85 | 143.2 | 135.8 |
| me-south-1 | 278.64 | 164.91 | 176.61 | 230.63 | 212.93 |
| as-east-1 | 4.65 | 117.16 | 127.43 | 178.69 | 175.61 |
| us-east-1 | 116.68 | 3.14 | 12.98 | 64.17 | 64.3 |
| us-east-2 | 128.35 | 16.07 | 4.88 | 52.22 | 59.38 |
| us-west-1 | 178.17 | 62.14 | 51.78 | 4.51 | 22.31 |
| us-west-2 | 175.06 | 62.7 | 53.59 | 22.25 | 3.69 |

Table A.4: AWS matrix - part 4.