UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO
UNIVERSIDADE FEDERAL DE SANTA CATARINA

Scalable and Resilient Byzantine Fault Tolerant Consensus

Ray Willy Neiheiser

Supervisors: Doctor Luís Eduardo Teixeira Rodrigues
Doctor Carlos Barros Montez
Co-Supervisor: Doctor Miguel Ângelo Marques de Matos

Thesis approved in public session to obtain the PhD Degree in Computer Science and Engineering

Jury final classification: Pass With Distinction

2022

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO
UNIVERSIDADE FEDERAL DE SANTA CATARINA

Scalable and Resilient Byzantine Fault Tolerant Consensus

Ray Willy Neiheiser

Supervisors:	Doctor Luís Eduardo Teixeira Rodrigues
	Doctor Carlos Barros Montez
Co-Supervisor:	Doctor Miguel Ângelo Marques de Matos

Thesis approved in public session to obtain the PhD Degree in Computer Science and Engineering

Jury final classification: Pass With Distinction

Jury:

Doctor Jean Everson Martina, Universidade Federal de Santa Catarina, Brazil
Doctor Rodrigo Seromenho Miragaia Rodrigues, Instituto Superior Técnico, Universidade de Lisboa
Doctor Miguel Nuno Dias Alves Pupo Correia, Instituto Superior Técnico, Universidade de Lisboa
Doctor Carlos Barros Montez, Universidade Federal de Santa Catarina, Brazil
Doctor Fabíola Gonçales Pereira Greve, Universidade Federal da Bahia, Brazil
Doctor Ittai Abraham, VMware Research, USA

Funding Institution:
Capes

2022

Scalable and Resilient Byzantine Fault Tolerant Consensus

to all the people that supported me

Acknowledgements

I want to thank all the amazing people that contributed to my research and accompanied me during these four years. First, I want to thank all the people advising me over the past years, Luís Rodrigues, Luciana Rech, Joni Fraga, Miguel Matos and Carlos Montez. I'm deeply grateful for their dedication to my professional and scientific development. A special thanks goes to Luís Rodrigues not only for encouraging me to go down this path, but also for all the constructive input, ideas and guidance he has provided me over these past years, without him none of this would've been possible. Similarly, I want to thank Luciana Rech that always had an open ear and was strongly dedicated to helping me pass all the necessary hurdles to pursue this path. I also want to thank Joni Fraga for taking me in initially and being an important anchorpoint in the early phase of my Ph.D, Miguel Matos, for his dedication, ideas and insightful feedback and finally Carlos Montez for supporting me in the final phase of this project.

I also want to thank all the people at INESC-ID and LAPESD for their insightful comments during the course of this project which not only helped to see our project from different angles but also helped me hone my presentation skills.

Thank also goes towards all the master and Ph.D. Students I encountered on my path, the conversations we had and their input and support. A special thanks in this context goes to Gustavo Inácio that worked diligently on a series of side-projects with me. In this context I also want to thank all people in the Open Source LDT-Team for their input and patience with me, all the dedicated researchers that reviewed my papers and provided insightful feedback and the general public that indirectly or directly provides the funds for all of our research.

I deeply thank my significant other for her continuous support and encouragement, always being there for me and being an important anchor on this journey and my parents, not only for their financial support when I needed it but also for always encouraging me to pursue whatever makes me happy.

Finally I want to thank Lau Cheuk Lung for opening me eyes for the world of research and Capes for financing my research.

"If a machine is expected to be infallible, it cannot also be intelligent." (Alan Turing)

Abstract

The increasing popularity of blockchains in addressing an expanding set of use cases, from enterprise to governmental applications, led to a growing interest in permissioned blockchains. In contrast to their permissionless counterparts, permissioned blockchains can ensure deterministic transaction finality which is a key requirement in many settings and can offer high throughput in small-sized systems. Some emerging use cases for permissioned blockchains require the system to scale to hundreds of participants. However, most permissioned blockchains are based on variants of classical byzantine fault-tolerant (BFT) consensus protocols that scale poorly with the number of participants. Such scalability limitations stem from bottlenecks both at the network and processing levels that result from the large number of messages that need to be sent, received and processed to reach consensus. Most attempts to solve this problem on a large scale end up reducing the overall resilience of the system and endanger both safety and liveness in the presence of byzantine failures. An approach that distributes the load equally among a set of processes are tree-based algorithms. However, due to the additional communication steps, tree-based approaches still display insufficient throughput on a geographic scale. Tree structures are also inherently more sensitive to byzantine faults, and constructing robust trees in the presence of failures is not a trivial matter. This thesis proposes new techniques that address these problems. We leverage extensive pipelining to achieve high throughput on a geographic scale independently of the depth of the tree and present a reconfiguration algorithm that is able to construct a robust tree in optimal steps in the presence of failures. Experimental results show that Kauri, a prototype that incorporates the proposed techniques, can efficiently execute consensus in settings with more than 500 participants and can achieve up to over 58 times the throughput of competing approaches.

Keywords: Byzantine Fault Tolerance, Distributed Ledger Technology, Blockchain, Consensus

Resumo

A utilização cada vez mais frequente de blockchains para desenvolver aplicações distribuídas de cariz comercial e governamental, tem aumentado o interesse nas blockchains fechadas, em que os nós necessitam de uma autorização prévia para participar no sistema, o que permite controlar o número e características dos participantes. Isto permite executar protocolos de consenso que garantem finalidade, isto é, em que o acordo não pode ser revertido. Estes protocolos conseguem também oferecer bom desempenho quando o número de participantes é reduzido (na ordem das dezenas). No entanto, é hoje possível encontrar aplicações baseadas em blockchain que precisam de suportar centenas de participantes. Infelizmente, a maioria das blockchain fechadas usam variantes do protocolo de consenso byzantino PBFT, que tem pouca capacidade de escala, devido à utilização de padrões de comunicação que obrigam um ou vários nós a enviar, receber e processar um elevado número de mensagens que cresce linear ou quadraticamente com o tamanho do sistema. As soluções anteriores para mitigar estes problems sofrem de várias limitações, comprometendo a segurança e/ou a disponibilidade do sistema na presença de faltas. Esta dissertação apresenta novas técnicas para abordar estes problemas, baseadas na construção de árvores que podem ser reconfiguradas num número óptimo de passos e que suportam a execução concorrente, em "pipeline", de múltiplas instâncias de consenso. Resultados experimentais mostram que um sistema que concretiza estas técnicas, denominado Kauri, consegue suportar de forma eficiente o consenso em grupos com mais de 500 participantes, oferecendo um débito 58 vezes superior ao de trabalhos anteriores.

Palavras-chave: Tolerancia a Faltas Bizantinas, Livro-Razão Distribuído, Blockchain, Consenso

Contents

List of Figures

List of Tables

1 Introduction

The term blockchain describes the immutable storage solution of Bitcoin [47] proposed by Satoshi Nakamoto in this context in 2008. As such, it is one of the essential building blocks allowing Bitcoin to solve the double-spending problem of digital assets without requiring trusted intermediaries. Due to its decentralized and open-source nature, this new technology gave rise to a wide variety of derivatives, and the term *blockchain* has since then been used as an umbrella term for its countless successors [71, 56, 14] with the joint goal of building better decentralized distributed systems.

In the scientific literature, these systems are commonly called "Distributed Ledgers", where an immutable storage solution (e.g., the blockchain) is usually combined with a byzantine fault-tolerant consensus algorithm which guarantees that all honest processes agree on the order of blocks of transactions to be persisted. This guarantees, as a result, that all honest participating processes may eventually agree on the same state.

While the initial proposal of Bitcoin relies on Proof of Work (PoW) as the consensus algorithm, a vast number of different solutions emerged since its inception, including, but not restricted to, Proof of Stake (PoS), Proof of Space, Proof of Burn, etc. These approaches can be described as "permissionless" consensus algorithms, as anyone may freely participate as long as they possess sufficient resources (CPU, Stake, Memory).

In the scientific literature, however, numerous approaches had been elaborated already since the early 1980s surrounding the byzantine generals problem [40]. In the context of distributed ledgers, these solutions are usually described as "permissioned" consensus algorithms, as processes have to go through some sorts of an approval process to participate in consensus. This is done to prevent Sybil attacks, where an attacker launches sufficient processes to overpower the system.

Nonetheless, independent of the type of system, most existing protocols face a serious bottleneck as they usually rely on all-to-all communication to establish consensus, requiring an increasing use of bandwidth for message propagation and computation power for signature verification with growing numbers of processes.

Thus, it is no surprise that, in practice, the projects with the highest potential throughput in the blockchain space restrict the number of processes participating in consensus to

avoid this bottleneck [9]. However, due to this, either resilience or deterministic finality is sacrificed.

However, the use-cases for permissionless as well as permissioned consensus protocols that allow scaling to several hundreds of participants are vast. On the permissioned side, e.g., the initial Diem whitepaper states that: "Our goal was to choose a protocol that would initially support at least 100 validators and would be able to evolve over time to support 500–1,000 validators" [3]. Similarly, a recent paper from IBM [65] further emphasizes the need for deployments that support large numbers of processes for platforms such as Corda [13]. In fact, over the past years, numerous works in the academic literature attempted to address this problem through a vast selection of different approaches [27, 35, 37, 26, 1].

In a similar manner, permissionless blockchains that currently rely on smaller committees could improve their security guarantees significantly. A good example of this are Delegated Proof of Stake (DPoS) blockchains where a small committee (20-30 processes) is elected by the stakeholders to execute the consensus in their name. This results in excellent throughput [9], but comes with an inherent risk, as if there is no majority of honest processes in the current committee, safety may not be guaranteed [41, 30]. Other committee-based solutions such as Stellar have similar problems. In fact, in a recent incident, the Stellar blockchain came to a halt for several hours as a significant percentage of the consensus processes went offline [66].

To solve this, several approaches attempt to build hierarchical systems to avoid this bottleneck. A famous example is HotStuff [1] which was initially developed to be used in the Libra (now Diem) blockchain project. HotStuff leverages a star-topology, where a single process receives and processes a quorum of signatures and then distributes the result to the remaining participants, significantly reducing the message complexity of the approach. However, a single process still has to compute and disseminate messages from all participants resulting in an inherent bottleneck.

Due to this, other proposals in the literature like Steward [2], Fireplug [51], or ResilientDB [58] create static hierarchical groups where intermediaries are selected to relay and pre-process information to avoid this bottleneck. However, due to their static nature, the number of faults they tolerate is limited, where, instead of global limits on the number of faults, there are local limits for faults depending on the place in the hierarchy (i.e., there can be no faulty majority in any of the groups).

Other recent protocols like Byzcoin [35] use a tree-topology to aggregate and process the signatures avoiding the before-mentioned bottleneck. However, this results in a significant trade-off as the additional number of communication steps increases the latency significantly, which may also affect the throughput negatively. Besides that, as faulty internal nodes in the tree might prevent consensus, the protocol has to fall back quickly to a star or clique in the presence of failures.

Therefore, there is a lack of proposals in the scientific literature and in practice that

avoid the bottleneck of star and clique-based approaches while maintaining high throughput and high resilience in geographically-distributed environments.

1.1 Objectives and Contributions

The general objective of this work was developing techniques to solve the above mentioned shortcomings of existing approaches.

In detail, we created flexible tree structures that distribute the communication and computation load fairly among the participating processes but are also able to withstand a large number of byzantine failures without requiring to fallback to a star or clique and without sacrificing safety in the presence of $f \leq \frac{N-1}{3}$ failures for a total of N processes.

More specifically, our requirements may be divided into:

- Reconfiguration of tree structures in optimal steps in the presence of failures;
- Compensation of high latency resulting from the tree structure in a geo-distributed tree environments;

We created a prototype called Kauri, including the proposed techniques, outperforming state-of-the-art approaches by a factor of up to 58 times the original performance. In the following, we describe each of the goals individually in detail.

1.1.1 Optimal Reconfiguration

The first and most vital step of making trees viable communication structures in the context of byzantine fault-tolerant consensus is the development of an efficient reconfiguration approach. Past solutions like Byzcoin [35] construct random trees and fall back to a clique if consensus does not terminate within some time limit. While Omniledger [37] and Motor [36] enhanced this process by instead slowly falling back to a star, multiple subsequent faulty leaders require substantial timeouts to be detected. Meanwhile, PBFT [16] or HotStuff [1] are always able to reconfigure to a valid structure in at most $f + 1$ steps in the worst case (f consecutive faulty leaders) which is optimal. However, while this is fairly trivial for cliques or Star topologies, this is significantly more complex in the case of trees. As this requires an exponential number of steps [36].

Based on this, we established a set of criteria for the reconfiguration algorithm.

- Optimal Reconfiguration until a certain threshold of failures f_a in the worst case.
- High probability of reaching a viable configuration in optimal steps in the presence of $f \leq \frac{N-1}{3}$ failures.
- Avoiding to fall back to a star topology.

The contributions of this thesis do not only entitle a reconfiguration algorithm fulfilling the above criteria but also a set of equations that allows assessing the exact number of failures f_a a given topology can tolerate before being unable to fulfill optimal reconfiguration. Besides that, we also describe a tree topology that is able to tolerate $f = \frac{N-1}{3}$ failures and prove that it is optimal.

1.1.2 Compensating Latency

In comparison to PBFT, HotStuff already requires twice the message round trip time to achieve consensus. Not only that, but HotStuff even requires one additional consensus phase before being able to agree on a given set of transactions [16, 1]. In the context of trees, this is even more severe, as increasingly deep trees extend the consensus latency significantly. While this would not be a problem in a local data center environment where round trip latencies are below one millisecond, in this thesis, we are considering a distributed ledger environment where processes are geographically distributed over the globe and network latencies may average somewhere between 100 and 200 milliseconds. Thus, the network latency has a significant impact on the throughput of the system. For this reason, HotStuff proposes a pipelining mechanism that allows, in each communication round, all consensus phases to be executed in parallel. As a result, after each communication round, one block is produced. This results in a significant increase in throughput, allowing HotStuff to display a similar throughput as PBFT even for small system sizes. However, as trees require additional communication steps, the pipelining in HotStuff is not sufficient to compensate for the additional communication overhead that is inherent to tree structures.

We, therefore, developed a pipelining technique similar to what is used in [68] which we adapted to the distributed ledger environment, allowing us to not only compensate the inherent tree latency but even outperform HotStuff by up to a factor of 58 by fully leveraging the communication and computation load distribution of the tree.

The contributions of this thesis, therefore, also entitle a novel pipelining technique in the context of distributed ledger consensus as well as a theoretical model that not only allows to reason about the potential speedup of the system but may also be used to configure the system in practice to achieve optimal throughput/latency levels.

1.1.3 Research History

This thesis was developed in the context of a Cotutelle agreement between the Department of Automation Engineering of the Federal University of Santa Catarina and the Informatic Engineering Department of the University of Lisbon.

Specifically, it was developed in the LAPESD (Distributed Computing Laboratory) at the Federal University of Santa Catarina and INESC-ID (Computer System Engineering Research and Development Institute) in Lisbon.

It can be treated as an extension of the work that was elaborated in the context of my master thesis, where a static hierarchical byzantine fault-tolerant architecture for distributed graph databases was developed. However, we noticed that these hierarchical structures are very inflexible and are subject to certain attack vector that endanger the liveness of the system.

Thus, in this thesis, we will treat more general, flexible hierarchical models in the presence of byzantine failures in the distributed ledger environment.

1.1.4 List of Publications

In the context of this Ph.D., a number of papers were published in both scientific journals as well as published and presented at scientific conferences. We divide the list of publications into two categories. First, the list of publications directly associated with the topic of this Ph.D., and second the papers that were published during this Ph.D., but are either related to secondary projects, student advisory, or are a continuation of my masters project.

Papers published related to the main topic:

- R. Neiheiser, L. Rodrigues, and M. Matos. "Kauri: Scalable BFT Consensus with PipelinedTree-Based Dissemination and Aggregation". In: *Proceedings of the 28th ACM Symposium on Operating Systems Principles*. SOSP '21. Online: Association for Computing Machinery, 2021
  This paper contains the main contributions we developed in the course of this thesis. As such, it presents our pipelining technique and the simplified version of the reconfiguration algorithm, as well as our theoretical model and a large percentage of the experimental results.

- R. Neiheiser, L. Rech, and J. da Silva Fraga. "Constantino: Uma Arquitetura BFT Escalável e Eficiente para Blockchains". In: *Anais do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuidos*. SBC. 2019, pp. 127–140
  This paper presents preliminary work. It focuses on the development of a hierarchical consensus algorithm. Nonetheless, in the further course of the thesis, we've further abstracted our techniques and, instead of developing our own consensus algorithm, applied our techniques to existing systems.

Secondary papers:

- R. Neiheiser, L. Rech, M. Bravo, L. Rodrigues, and M. Correia. "Fireplug: Efficient and Robust Geo-Replication of Graph Databases". In: *IEEE Transactions on Parallel and Distributed Systems* 31.8 (2020), pp. 1942–1953
  Extensive evaluation of my Master's prototype including additional functionalities.
- R. Neiheiser, G. Inácio, L. Rech, and J. Fraga. "HRM Smart Contracts on the Blockchain: Emulated vs Native". In: *Cluster Computing* (2020)
  Extension of the Paper treating decentralized human resource management. Compares the differences of deploying similar projects with the help of smart contracts or as layer-2 solutions.
- M. Bravo, L. Rodrigues, R. Neiheiser, and L. Rech. "Policy-Based Adaptation of a Byzantine Fault Tolerant Distributed Graph Database". In: *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. 2018, pp. 61–71
  Adaptive extension of the prototype developed during my Master's.
- R. Neiheiser, G. Inácio, L. Rech, and J. Fraga. "HRM Smart Contracts on the Blockchain". In: *2019 IEEE Symposium on Computers and Communications (ISCC)*. 2019
  Creation of a decentralized process for human resource management.

1.2 Thesis Structure

The remainder of the text is organized as follows. Chapter 2 presents concepts of Fault Tolerance, Distributed Systems, Security, and Distributed Ledgers. Then, Chapter 3 discusses the related work in consensus present in the literature of byzantine fault tolerance. Next, in the main part of this document, Chapter 4 presents the proposal of the work that was developed during this doctoral degree. The following Chapter presents the obtained experimental results, and finally, Chapter 6 concludes the thesis.

2

Background

This chapter elaborates on the basic principles of building robust fault-tolerant distributed systems. First, we discuss basic cryptographic primitives, an essential building block to construct dependable systems. We continue with the basic principles of dependability and the possible threats a system might face, and how systems can be built to cope with this. Finally, distributed ledgers and their underlying technology are discussed in-depth.

2.1 Cryptographic Principles

In a system where two or more parties exchange messages and we expect interference of malicious actors, there are three essential properties the underlying communication principle ought to fulfill: Integrity, Authenticity, and, optionally, Confidentiality. Integrity encompasses guaranteeing that the message that was sent by some process p_i also arrives at another process p_j without alterations (i.e., was not tampered with in the process) or else the alteration of the message is detected. Authenticity, on the other hand, is about allowing process p_i to identify process p_j as the author of a given message (i.e., a process is not able to impersonate others). Finally, confidentiality attempts to guarantee that no other party is able to intercept and read the messages passed by p_i to any p_j [67].

This is where cryptography comes into play. In this section, we present a number of cryptographic principles that are widely used to fulfill the above properties and present the cryptographic algorithms that are discussed or leveraged in the context of this thesis. Note, as this thesis was developed in the context of distributed consensus in blockchain environments, we have a special focus on integrity and authenticity.

2.1.1 Symmetric and Asymmetric Cryptography

The roots of cryptography go back far in history, long before the invention of modern computers (e.g., encrypted messages during the roman empire). Cryptography is usually divided into symmetric cryptography, where a single common key is used for encryption and decryption (both communication partners hold the same key), and asymmetric

cryptography, where there is one key for encryption, and another key for decryption (the communication partners hold different keys) [67].

Symmetric Cryptography

In symmetric cryptography, two parties exchange a single key in a secure way and may then use this key in order to establish a confidential channel. While this is very efficient, on a large scale, it eventually becomes unfeasible as all processes have to exchange unique keys. In addition to that, not only does the secure exchange of a key between two parties require a complex protocol (usually involving asymmetric cryptography), but either party might also leak the secure key breaking the security guarantees of the system. As such, symmetric cryptography requires some form of trust between the two parties.

Asymmetric Cryptography

Asymmetric cryptography requires two keys, typically a public key (publicly known) and a private key (only known by its owner). A process p_j that wants to send an encrypted message to a process p_i may simply encrypt a message with the public key pub_i of p_i such that only process p_i may decrypt the message again (with the help of their private key $priv_i$). For a similar level of security, asymmetric cryptography is asymptotically more computationally intensive than symmetric cryptography. However, as opposed to symmetric cryptography, there is no shared secret among a set of processes.

Elliptic Curve In the context of asymmetric signatures we also have to talk about Elliptic Curve algorithms that surged in importance in recent years due to the growth of the distributed ledger technology. Elliptic Curves are a class of efficient algorithms that construct asymmetric signatures that are not only verified significantly faster in many instances than classic asymmetric signatures but are often also able to generate relatively short signatures in comparison, which is very important as this significantly reduces the storage overhead.

2.1.2 Cryptographic Hash Functions

One of the fundamental building blocks of many distributed systems are cryptographic hash functions. A hash function H is a function which maps an input of arbitrary length to a fixed sized output. A robust hash function has to fulfill the following three basic properties [59]:

- Efficiency: Ease of computing $H(v)$ for any input v.
- Collision Resistance: Difficulty of finding two inputs v and v' with the same Hash $H(v) = H(v')$
- Unidirectional: Difficulty of finding the input v given the Hash $H(v)$.

In a nutshell, hash functions map an input from an infinite domain (arbitrary length) to a finite domain (fixed length) in a deterministic way. As such, given any input v and its hash $H(v)$, we are able to detect any alterations of v by re-executing the hash function and comparing it to the previously computed hash.

However, this alone does not guarantee integrity, as, when a given value v is sent alongside its hash $H(v)$ to a recipient, both v and $H(v)$ could be altered by a malicious party.

2.1.3 Digital Signatures

Digital signatures ensure that a process p_j can identify the creator p_i of a given message ms_i. It again includes a key pair consisting of a public and a private key (pub_i and $priv_i$). The private key is only known by the creator of the signature, and the public key is publicly available. Its functionality is defined using two primitives:

- SIGN($priv_i$, v): Applies the private key $priv_i$ on a given value v. This produces the signature sig_v
- VERIFY(pub_i, v, sig_v). Is used to verify the signature sig_v resulting in a boolean value (true or false) depending on the correctness of the signature. It returns true if and only if, when computing the public key pub_i over the signature sig_v, value v is returned.

In this scheme, it is important that it is computationally unfeasible to obtain either a valid signature sig_v without knowing the private key $priv_i$ or to obtain the private key from the signature sig_v and the public key pub_i. In the context of this work, we denote a signed message ms by process p_i as ms_i [67].

As such, with the help of digital signatures, authenticity is fulfilled, as only p_i is able to construct a given signature using $priv_i$. In addition to that, integrity is guaranteed, as even if a malicious process was able to change the value v the signature sig_v would not match anymore, and the alteration is easily detected.

2.1.4 Message Authentication Codes

Message Authentication Codes (MACs) are used to authenticate messages that are exchanged between parties that share a common secret key (symmetric cryptography) [28]. MACs provide integrity and authenticity and are widely studied in the literature due to their speed advantages compared to signatures using asymmetric cryptography. However, in certain protocols, when messages are broadcast or relayed, the number of MACs that have to be appended to each message grows linearly with the system size [17, 1].

2.1.5 Signature Schemes

A wide selection of different signature schemes exist. Especially notable in the context of this work are Threshold Signatures and Multi-Signatures.

Threshold Cryptography

Threshold Cryptography is also known as k out of n cryptography as only a subset of signatures k of all participants n is required to sign or verify (or encrypt/decrypt) a message. Following this scheme, each process of a given group g owns a partial key $psig_m$ that may be used to sign a given message m. After signing, any server can combine the partial keys (as long as in possession of at least k partial keys) to obtain the final threshold signature. This signature may then be verified in $\mathcal{O}(1)$ complexity by any process.

However, *Threshold Cryptography*, while being highly efficient as only one final signature has to be sent and verified, requires a lengthy and complex protocol to distribute the partial keys every time the set of participants changes. In addition to that, while threshold signatures may be partially aggregated and partial aggregates combined in a multi-step protocol, partial aggregates may not be verified as at least k signatures are necessary for verification [22].

Multi-Signatures

Multi-Signatures are similar to *Threshold Signatures*, as they allow aggregating multiple signatures into a single signature which can then be verified in $\mathcal{O}(1)$ steps. However, compared to threshold signatures, multi-signature schemes are more flexible as they allow verifying which and how many processes contributed their signature to a given multi-signature.

There are two main types of multi-signatures. The first one is based on *Schnorr* Signatures [61]. These signatures require $\mathcal{O}(1)$ storage and can be verified in $\mathcal{O}(1)$ steps. However, they require an interactive protocol to construct one specific multi-signature which requires multiple rounds of communication.

BLS signatures [11] do not necessarily require an interactive protocol and may be aggregated at any process. However, they are significantly more complex computationally and require a vector of participant identifiers to be transferred alongside the signature.

In the context of this work we will use the non-interactive *bls12-381* multi-signatures proposed in [10].

Comparison

As in this work we'll be comparing different algorithms that rely on different cryptographic schemes, we evaluated the computational complexity of the ones that are important in the context of this paper.

- *secp256k1* (Elliptic Curve Algorithm, e.g. Used in Bitcoin [47] and HotStuff [1]);

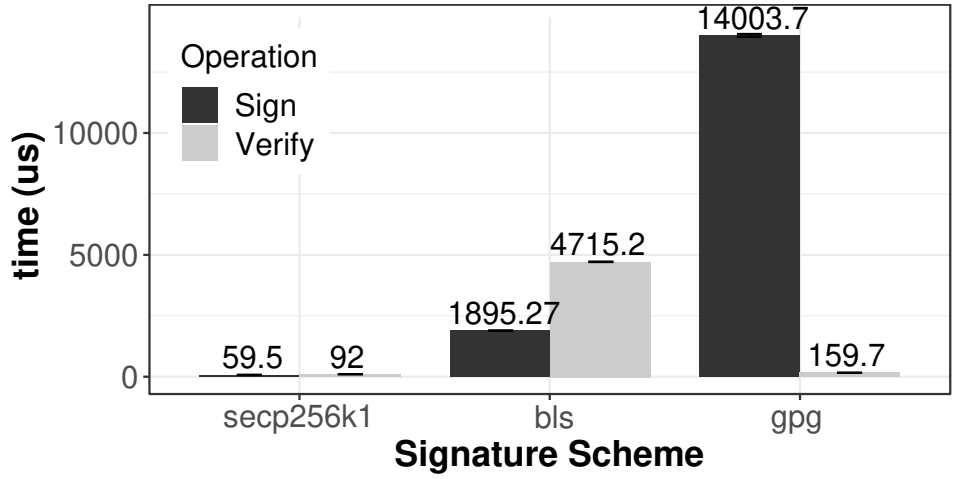

Figure 2.1: Cost of cryptographic operations (Sign and Verify)

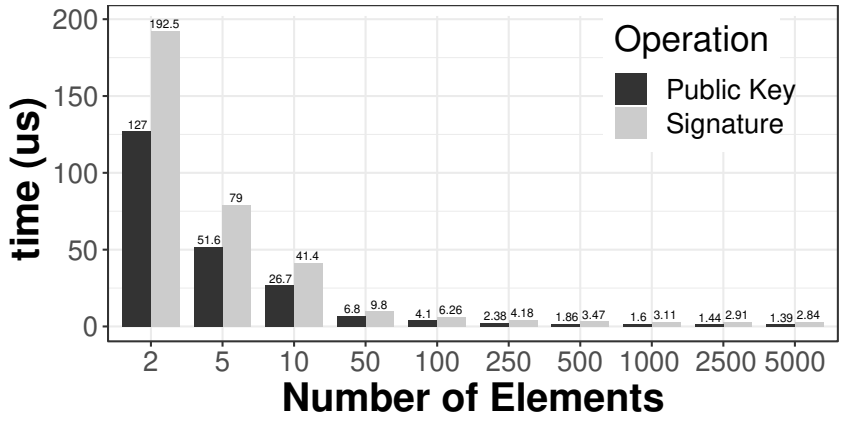

Figure 2.2: Cost of cryptographic operations (BLS Aggregate)

- *bls12-381* (BLS multi-signature - e.g. used in Chia [20]);
- *gpg* (e.g. used in most Linux distributions)

Figure 2.1 shows the cost of a single sign and verify operation averaged over 100 runs for each scheme. This was run on a machine with an Intel i7-8750H CPU andt 32Gb RAM. Quite visibly, *gpg* is substantially slower than *secp256k1*, where the latter also clearly outperforms BLS. However, while the cost of *gpg* and *secp256k1* grows linearly with the number of signatures, *bls* amortizes this cost through aggregation. The results for BLS aggregation are shown in Figure 2.2 and can be decomposed into the aggregation of multiple signatures to create one single aggregated signature and the aggregation of public keys to create an aggregated public key that is required to verify aggregated signatures. Interestingly the cost of aggregation decreases quickly with the number of elements (number of signatures or public keys) while the verification cost remains constant (as cryptographic pairings can be cached). In contrast, the verification cost of the

other schemes grows linear with the number of signatures. This means that after ≈ 55 signatures (*i.e.* input of processes), the overall cost of *secp256k1* surpasses that of BLS.

2.2 Dependability

Dependability can be understood as the branch in computer science that is related to upholding certain guarantees or attributes that are associated with a given service or system. As such, in the field of dependability, threats to the desired state of a system are identified, and techniques have to be developed to cope with different levels of threats [6]. Fundamentally, a dependable system is a system that can be trusted to be able to deliver a given quality of service.

A dependable system has to fulfill the following properties [6]:

- Availability: The ability of the system to answer a request.
- Reliability: The continuous delivery of correct service without interruption;
- Safety: The avoidance of negative consequences in the presence of failures;
- Integrity: The protection of the system against unauthorized alterations;
- Maintainability: The ease of restoring the system to a correct state.

These properties can be condensed into two basic properties *Safety* and *Liveness*. Safety determines that a program follows its specification (producing the correct output), and liveness is usually associated with the termination of a program or algorithm (continuously producing output) [39].

2.2.1 Threats

In the context of dependable systems, threats are usually discussed in the context of faults, where a fault might be the reason for a system to deviate from its specification. Such a fault might originate from a diverse set of factors, ranging from weaknesses or flaws of the code of the service, the underlying system, its configuration or operation (internal causes) or might have been introduced by an external actor, like given environmental conditions (temperature, weather) or even caused by an intruder (external causes).

The manifestation of such a fault is usually described as an error, which itself might result in the failure of the system leading to a deviation of the specification of the given service [6].

As a simple example, the fault might be a missing condition in the code forgotten by the developer, the error is the reaction of the system when encountering the missing condition, and the failure is the shutdown of the service due to the fault. As the service is unable to react to queries in this state, it is deviating from its specification. Thus, the

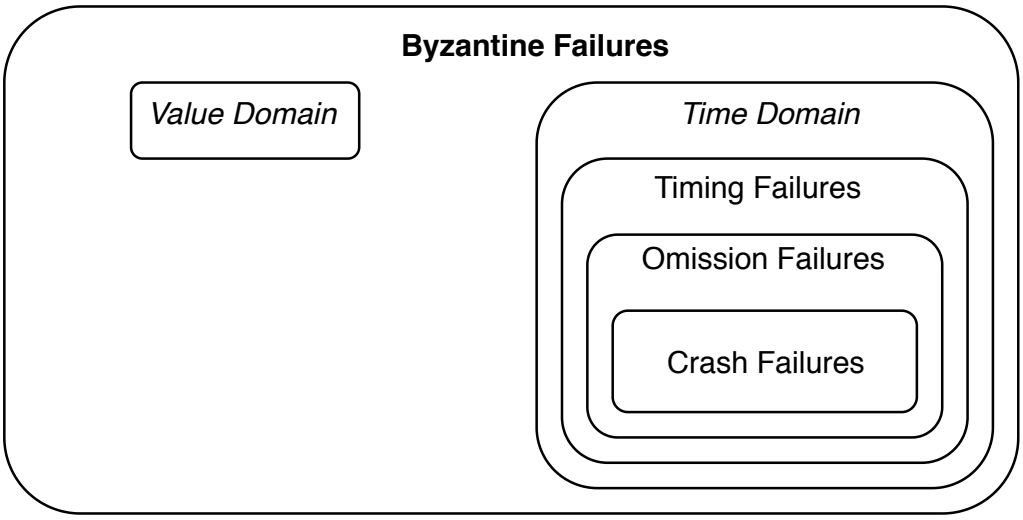

Figure 2.3: Fault classification based on failure semantics [21]

failures resulting from a fault may impair the safety and liveness properties of the whole system.

In the context of this work, we are specifically targeting distributed systems. Thus, we do not only have to deal with failures related to the execution of the algorithm or program but also with potential failures of the underlying communication channels. In addition to that, while there are numerous ways to classify failures in the literature (based on the origin of the fault, persistence, etc.), in the context of this work, we have a special interest in the impact of the failure. This is usually described as *failure semantics* and identifies different types of faults that might surface during the runtime of a system and the potential impacts of the failures they may cause.

Classically, failures are divided into two domains, the time domain (e.g., service is interrupted) and value domain (e.g., incorrect values are returned) where the failures are further divided into four different categories attending different domains [21].

- Crash failures: The complete interruption of the system or service, e.g., due to a crash. In this state, the system or service will not attend to any requests until a valid state is restored (time).

- Omission failures: The failure of the system to attend to certain requests while potentially attending others, e.g., responding to some users but not others (time).

- Timing failures: The failure of the system to respond to a given request within a certain time interval (time).

- Arbitrary failures: The system produces arbitrary values or arbitrarily only responds to certain requests. Often also described as byzantine failures, which include malicious faults (time and value).

Figure 2.3 visualizes the classification of the cited faults, showing the relation between the different types of faults, ranging from the most restrictive type of failure (crash) to the most general failure (byzantine). Failures in the time domain can be detected relatively easily through timeouts. Byzantine failures, however, are much more difficult to deal with as they might impact both the time and value domains. This malicious behavior is difficult to detect as it requires knowledge of the application and its semantics.

As we are dealing with distributed systems, there are multiple system components that might fail. We describe the encapsulation of one such component as a process. We define a correct process as a process that responds to a correct request with the expected reply (following its specification).

As we are dealing with distributed ledgers, we assume the byzantine failure model. Thus, the fault tolerance techniques that are discussed in the following sections are especially focused on this type of failure.

2.2.2 Methods

In order for a dependable system to follow its specification, it is necessary that, throughout the planning phase up to production, certain methods, techniques and tools are used.

In [6] a set of methods and techniques are identified which aid at achieving a dependable system. These are: *Fault Prevention*, *Fault Removal*, *Fault Forecasting* and *Fault Tolerance*. *Fault Prevention* and *Fault Removal* are focused on the usage of methodologies, tools, and tests during the development of the project, minimizing the occurrence of faults. *Fault Forecasting*, relies on analytic methods (often based on stochastic processes) and trys to foresee the system's behavior during its life-cycle, trying to estimate the reliability and availability of the system itself.

Since the techniques to prevent and remove faults are not exhaustive and also cannot cover the entire life cycle of a project, it is necessary to use mechanisms such as *Fault Tolerance* to guarantee the continuous availability of a system or service. These mechanisms are often based on redundancy (time/software/physical) which allow the system to evolve following its specifications even in the presence of failures and intruders in the system.

Another important factor besides redundancy is diversity, which is strongly related to N-version Programming (code made by independent teams, using different programming languages, methodologies, etc.) to shield the system against similar faults on different processes of a service. In other words, diversity decreases the occurrence of correlated faults. This makes it less likely to have several processes suffer from the same failure [5].

2.2.3 Fault Tolerance Techniques

There is a wide selection of techniques that can be used to tolerate faults. These techniques can be divided primarily into two groups: *Error Detection and Recovery*, and *Masking of Faults*. Error detection could, e.g. be done with the help of timeouts, recovery could

be the attempt to restore the system to the last valid system state (i.e. snapshots) and masking errors could be done with the help of redundancies (e.g. replication). In practice, usually a combination of both approaches is used. Replication is applied to maintain operation in the presence of failures, and failure detection and recovery are then applied to recover the system to the previous state [6].

Replication may also be active or passive, where active replication increases the set of active participants and passive replication adds a set of processes that do not engage with the system until failures are detected. In the context of this thesis, we focus on active replication, also known as *State Machine Replication* [6].

State Machine Replication

In State Machine Replication (SMR), the system state is represented by *State Variables* which may be transformed through a set of *Commands* where *Commands* are mandatorily deterministic. This is necessary to guarantee that a set of processes, given a set of *State Variables* and an ordered list of *Commands* generates the same result on each process [6]. This requirement is also called determinism of replicas [60][1].

According to Schneider, the progress of a *State machine* depends on the fulfillment of the following two properties:

- Agreement: All correct replicas receive the same requests.
- Order: All correct replicas process the received requests in the same order.

The requirements of agreement and order are typically fulfilled by consensus protocols [31], where consensus is used to have the set of processes agree on a given value or order of values, persist the new state and notify the client. A client accepts the result of the operation after receiving $f + 1$ equal confirmations, where f is the maximum number of faulty processes.

Consensus

Consensus is one of the fundamental problems of distributed systems. This is the case since as soon as data and or processing is distributed over a set of physical machines, some sort of mechanism is necessary to have the different processes agree on a given state.

More formally, it consists of two primitives [31]:

- propose(G, v): Proposing the value v to a set of processes G.
- decide(v): Notifying the processes about the decided value v.

[1] Replicas which start from the same initial state going through the same sequence of requests in the same order have to come to the same result

In order to satisfy *Safety* and *Liveness* these primitives have to fulfill the following properties [15] (p. 245):

- "Agreement: No two correct processes decide differently."
- "Termination: Every correct process eventually decides some value."
- "Integrity: No correct process decides twice."

In addition to the three properties above, *Validity* also has to be guaranteed. In [15] (p. 246) the two following definitions of validity are discussed:

- "Weak Validity: If all processes are correct and propose the same value v, then no correct process decides a value different from v; furthermore, if all processes are correct and some process decides v, then v was proposed by some process".
- "Strong Validity: If all correct processes propose the same value v, then no correct process decides a value different from v; otherwise, a correct process may only decide a value that was proposed by some correct process or the special value $\perp$".

In the context of this Thesis we consider *Weak Validity*. While malicious processes may propose blocks with spurious data, in the context of blockchain based systems, transactions require the matching client signatures to affect the system state.

While *Safety* depends on the fulfillment of *Agreement*, *Integrity* and *Validity*, *Termination* guarantees *Liveness*.

Thus, an algorithm that solves consensus must fulfill these four properties.

2.3 Distributed Ledgers

Bitcoin [47] in 2008, solved the double-spending problem of digital currencies with the help of a combination of the blockchain for immutable storage and a novel consensus protocol to synchronize state. The double-spending problem describes the difficulty of tracking digital assets without relying on central trusted components. In digital payment systems like PayPal, where a central agency manages a ledger with incoming and outgoing transactions of every user, it is pretty simple to track and control the flux of transactions to avoid double-spending. As such, if a user issues a digital currency transaction to another user, a central component can easily adjust their state such that if the user attempts to issue the same transaction to another user shortly after, the system can detect and prevent this easily. However, central entities, on the other hand, may censor or delay transactions or even steal money and therefore require trust to operate.

Meanwhile, while censorship in a decentralized system is significantly less likely, preventing double-spending is challenging. i.e., users may send transactions with diverging states to different processes simultaneously, the processes approve different transactions, and both receivers believe they received the assets accordingly.

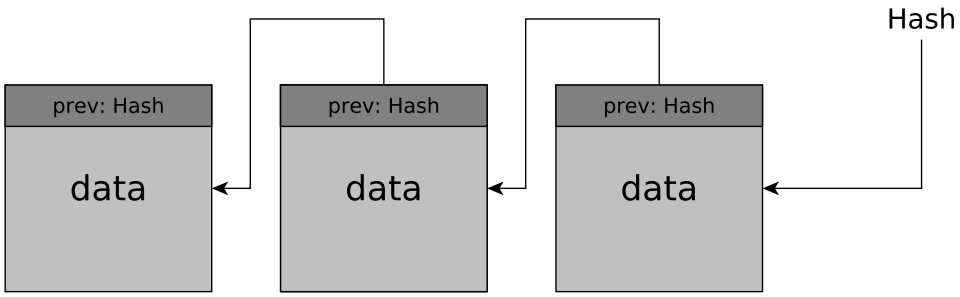

Figure 2.4: Blockchain Data Structure

This section focuses on the basic concepts of distributed ledgers, how they guarantee immutability and how they achieve consensus and prevent double-spending.

2.3.1 Blockchain

While the blockchain has long been used as an umbrella term for systems that are similar to Bitcoin, due to the creation of adjacent data structures with similar but distinct properties, the scientific community shifted to using *distributed ledger* as a more all englobing term.

The term blockchain explicitly describes the type of data storage used in Bitcoin and similar systems. In this work, we will solely focus on distributed ledgers that leverage the blockchain as their storage technology which is arguably the most common implementation.

The blockchain (depicted in Figure 2.4), roughly simplified, is a linked list of blocks where each subsequent block holds a hash of the previous block as a reference. This approach makes the entire chain immutable, as changing a block in any position would require updating every subsequent block. Thus, the blockchain relies on cryptographic hash functions allowing anyone to easily cross-check the integrity of a given block on the chain by calculating the hash of a given block and comparing it to the hash stored in the subsequent block in the chain. Thus, if anyone tampered with the data, the newly calculated hash and the hash stored in the subsequent block would differ, exposing the alteration (Section 2.1.2). This way, with the continuous growth of the blockchain, it becomes gradually more challenging to alter past data.

In order to calculate an up to date state of the system, the entire list of blocks has to be traversed, and the transactions in each block executed sequentially. In order to compare the state of a set of different servers efficiently, Bitcoin [47] uses Merkle Trees. Merkle Trees are another hash-based data structure where each leaf represents a given block, and the internal nodes connecting a given pair of leaves are their aggregate. The intriguing part of Merkle trees is that given the Merkle tree root hash and a block hash, it is possible to verify if the block is part of this tree highly efficiently. Due to this, a new

participant may download the blockchain of any other participant and then efficiently assert the correctness of the data by computing the Merkle tree of the local data, querying the Merkle tree root hashes of a sufficiently large set of other processes, and comparing it to the locally computed Merkle tree root.

In most distributed ledgers, each client is identified using a pair of asymmetric keys where the public key serves as their public identifier, and the private key allows the user to authenticate and issue transactions with the help of digital signatures in the system.

Building blockchains in decentralized peer-to-peer networks without trusted elements that are aware of all transactions is a complex task as it requires regular synchronization between the different peers [47]. Therefore, the primary tool for building such a distributed ledger is consensus, discussed in detail in the next chapter. Roughly simplified, with the help of consensus, all peers in the system may eventually agree on the system's state (the order in which transactions shall be persisted). Due to this, decentralized systems come with a significant performance drawback, as all participants have to agree on every single value. As such, to guarantee high scalability in terms of consensus participants and to deal with regular fluctuations of validators, several novel algorithms have been designed, which are explained in the following.

2.3.2 Blockchain Consensus

Distributed Ledgers can be divided into *permissionless* and *permissioned* ledgers. In permissionless ledgers, any server may join and leave at will. As such, it functions as an open system. Permissioned ledgers, on the other hand, can be divided into consortium ledgers, where a consortium of companies controls the existing nodes of the ledger, and private ledgers where the entire system is controlled by a single entity [70]. Permissioned ledgers can easily run traditional consensus algorithms due to their more restrictive nature, where horizontal scalability is still important as many use-cases of this technology include a potentially large group of participants. Meanwhile, permissionless ledgers require other approaches due to the openness of the system. While traditional consensus operates with the principle "one process one vote", which implies that every process has an equal say in the system, in permissionless systems, an adversary could easily create sufficient instances to overthrow the existing consensus (usually described as Sybil attack).

For this reason, novel algorithms were developed that allow horizontal scalability while being resilient against Sybil attacks. Due to the large number of participants, consensus using an algorithm that requires message exchange between all participants is very costly, especially considering wide area networks. Most blockchains, therefore, run an algorithm that, strongly simplified, elects a single process each round to decide on a block of transactions. Then, the elected process broadcasts their proposal, and the remaining processes verify and append it to their blockchain if valid [70].

The first algorithm that was developed in this context, initially proposed by Nakamoto [47],

was *Proof of Work* (PoW), where the participants (called miners) have to solve a computationally complex puzzle to participate in consensus. This way, the "one process one vote" policy is effectively transformed into "one CPU one vote", effectively increasing the resource requirement to overthrow the network significantly.

This puzzle usually involves finding a hash that matches a specific pattern (e.g., a hash that ends with the symbols 123). To solve this, a unique field in the block called *nonce* is altered until the hash matches the criteria. This approach also allows adjusting the difficulty dynamically by adapting the hash requirement (instead of requiring a hash ending with the pattern 123, the difficulty could be reduced by only requiring it to end in 23). The first process to solve the puzzle receives a reward, as an incentive, to participate honestly in the system. Therefore a part of the correctness of the system depends on game theory where participants are rewarded such that honest participation in the system is supposed to be more rewarding than dishonest participation. In addition to that, due to the high computational cost, the immutability of the blockchain is strengthened, as recalculating a chain of blocks would entitle solving the cryptographic puzzle for each of the blocks all over again.

Nevertheless, this results in an optimistic execution of consensus, as the fastest process to solve the puzzle will distribute its block to others. As such, it is relatively common for two processes to solve the puzzle at a similar moment and, due to the geo-distribution, deliver their block to different participants in a different order. This event creates a *fork* as the local blockchain of different participants differs for a certain period. This inconsistency is solved by the following process that solves the puzzle. As the next process has a specific local view of the state (accepted one of the two blocks), when it distributes its new block to all processes in the system, the other processes will override their current state due to the "longest chain rules" directive.

All participating servers have to execute this CPU-intensive algorithm resulting in a very high computational cost overall that is very wasteful [70]. For this reason, more efficient algorithms based on game theory have been created. One of them is *Proof of Stake* (PoS), where each server has to lock a certain quantity of cryptocurrency (stake) to participate (could also be described as a security deposit). This way, the miner is highly interested in the success and thus in the stability of the chain, as their stake could else lose value significantly [34]. A deterministic schedule then selects the following process to produce a block based on their stake. Nonetheless, as the participants with the most significant stake acquire even more stake, this eventually leads to a centralization of power. Not only that, but as existing stakeholders hold large quantities of the stake, it gets increasingly difficult for outsiders to enter the system.

Besides Proof of Stake and Proof of Work, other models exist like *Delegated Proof of Stake* (DPoS) where the stakeholders can elect a limited group of consensus nodes (witnesses) that participate in the consensus on their behalf. This approach reduces the number of validators participating in the consensus and improves the performance significantly [14]. Among the most efficient blockchains in terms of number of transactions

we find many *DPoS* chains as Hive [41], EOS [30] or Bitshares [62].

Outside of the named systems, other less popular models exist, as Proof of Elapsed Time [18] where Intel-based CPUs can prove having been idle for a particular period, Proof of Capacity [72] which is based on disk space, Proof of Burn [45] which is based on destroying cryptocurrencies in the consensus process, etc.

While there is a vast selection of different approaches, they have plenty in common as most of these systems, in practice, either rely on all-to-all communication and end up with crippled throughput or reduce the number of participants but endanger the safety of the system.

Consensus algorithms for permissioned blockchains, which are the main focus of this thesis, are discussed in detail in the next chapter.

2.4 Summary

This chapter introduced the fundamentals of cryptography and dependability necessary to understand the subject. As our work is centered around distributed ledger technology, this also included an explanation of the basic principles surrounding this technology.

In the next chapter, we discuss approaches from the scientific literature regarding byzantine fault-tolerant consensus. While most of these approaches assume permissioned models, it is possible to combine those with one of the above-mentioned permissionless solutions to create hybrid approaches that are both able to deliver specific system properties while also working in a permissionless environment [70].

3

Related Work

The problem of byzantine agreement was discussed initially in the context of synchronous systems [40], resulting in a recursive algorithm with a factorial message complexity. The first solution with a practical message complexity was *PBFT* [16] that, until today, serves as an inspiration for most byzantine fault-tolerant consensus protocols. *PBFT*, similarly to most modern approaches, provides safety in a completely asynchronous setting but requires synchronous phases to guarantee progress.

This is a necessary assumption since, as proven in [25], it is impossible for a consensus protocol to fulfill both liveness and safety in the presence of asynchronous network conditions. This is known as the famous *FLP* impossibility theorem. As a result, the concept of partially synchronous protocols emerged [23]. While, in this model, periods of instability may occur where messages might be delayed arbitrarily (asynchronous network conditions), there is an unknown Global Stabilization Time (GST) and a known worst-case network latency Δ where, after GST, messages arrive within Δ. Thus, in this model, safety is always guaranteed, while liveness is only established after GST.

While there are protocols like Honeybadger [44] that offer guaranteed liveness even under asynchronous network conditions, they do not provide deterministic safety (i.e., these protocols only provide safety at a "very high" probability). In addition to that, this group of protocols requires a substantial network complexity ($\mathcal{O}(N^4)$). Thus, due to our concerns for scalability and high throughput, we did not include this group of protocols in our analysis.

3.1 Classification of Approaches

We generally assume leader-based approaches, where a single process proposes a value for each round of consensus, which all other processes will then agree on. In this context, we classify the existing approaches into subgroups based on the communication pattern that is used by the leader to disseminate the proposal and by the remaining process to exchange their agreement.

- *All-to-All:* Leader process broadcasts block, all processes broadcast their vote and

verify and collect votes of all other processes.

- *Star:* Leader process broadcasts block. All processes send their vote to the leader that collects and verifies the votes and relays them in a batch once reaching a majority.
- *Gossip:* Leader process propagates block through gossip. All processes receive, verify, aggregate, and relay aggregates through multiple rounds of gossip.
- *Committee:* A small subset of processes is randomly selected. The leader sends the proposal to all processes in this committee, committee members exchange their vote internally and verify and collect votes of all other processes within the committee. After the end of consensus, each committee member distributes the result to the remaining processes before a new committee is elected.
- *Hierarchical:* First, the leader process disseminates a message to a set of cluster primaries. Next, cluster primaries verify and relay the message to their cluster members. Then, all processes send their vote to their respective primary, the primaries aggregate, verify and then exchange their respective aggregates with the other primaries. Finally, after constructing an aggregate consisting of sufficient votes, each primary relays this to their cluster members and persist the result.
- *Tree:* Leader process sends a value to their child processes in the tree, child processes relay to their respective child processes until reaching the leaf nodes. Next, all processes verify, aggregate, and relay the votes up the tree back to the leader. Finally, the leader collects the last aggregate and then restarts the distribution process.

3.2 Byzantine Fault Tolerant Consensus

In the following, we present the most dominant proposals based on the classification above.

3.2.1 All-to-All Protocols

The first and most prevalent group of protocols is based on *all-to-all* communication. Thus, the communication cost of these protocols grows quadratic with the number of participants. Most of these protocols are direct derivatives of *PBFT* [16] and use the communication pattern depicted in Figure 3.1. In this protocol, there are multiple steps, server processes P (P_0...P_3) and one client process C.

PBFT is a leader-based protocol where one of the participants fulfills the unique role of the *leader* (in this case P_0). The protocol starts with the client sending transactions to the leader. If the leader is correct and is not suspected to be faulty, it will propose a value as input to consensus in the *pre-prepare* phase. This may either be a single transaction or a batch of transactions (i.e. a block).

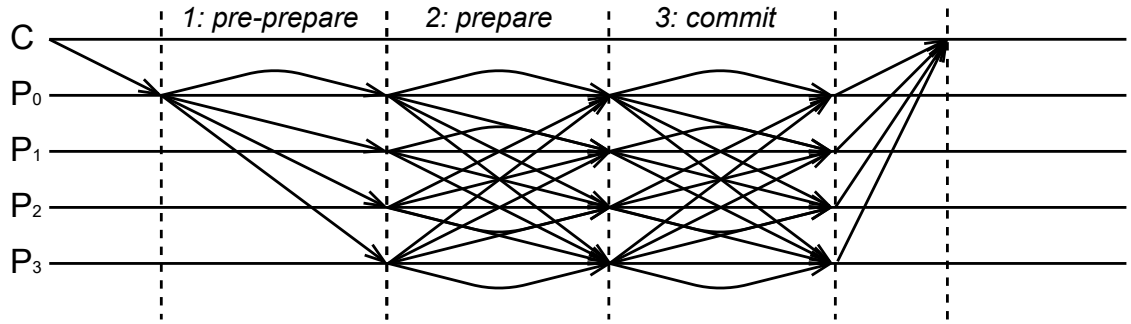

Figure 3.1: Three Phase Protocol - PBFT

Next, each process verifies the proposal of the leader and, if the proposal follows the previously agreed-upon rules, each process broadcasts their vote with the proposed value in the *prepare* phase.

At the end of this phase, each process verifies the received *prepare* messages, and if a full byzantine quorum was collected ($2f + 1$ *prepare* messages), it locks the value and broadcasts a *commit* message to notify the other processes.

After receiving $2f + 1$ *commit* messages, each process can be certain that the majority of processes (at least $f + 1$ honest processes) did receive a full quorum in the *prepare* phase. They persist the agreed-upon value and notify the client about the result.

If any of these phases fails (i.e., a timeout is reached), a new leader is selected through a view-change protocol. The view change protocol works very similarly to the consensus protocol. After the timeout is reached, each process determines the next leader deterministically (i.e., rotating the leader in round-robin fashion) and broadcasts the selection in a view-change message. After receiving $f + 1$ view-change messages, any process that did not run into the timeout yet also determines a new leader and broadcasts its decision. This way, correct processes that were lagging behind may catch up to the remaining processes. Finally, after receiving a byzantine quorum of view-change messages ($2f + 1$), the elected process assumes the role of the leader and restarts the protocol. This algorithm results in a cost of $\mathcal{O}(n^2)$ for each view-change, and, as there is a possibility of f subsequent leaders, it might have to be repeated f times. As such, in the worst case, this results in a cost of $\mathcal{O}(n^3)$.

SBFT [27] is an optimization of *PBFT* where each process creates a single aggregated signature. Therefore, the processes only have to broadcast and verify a single signature each round instead of the whole set of signatures. This strategy significantly reduces the bandwidth load of the subsequent rounds and results in a lower computational burden on all processes.

However, based on the above algorithm, in both *SBFT* and *PBFT*, the leader clearly has more work than the remaining participants, as it needs to collect the requests from clients, broadcast the batch of transactions, and also run the same protocol as the remaining

participants. One way to alleviate the load on the leader is by rotating the role of the leader among all participants in every consensus instance. Another reason to rotate the leader regularly is to reduce the potential influence of a faulty leader process. *Spinning* [69] is a protocol that uses this strategy. As an extension of Spinning, *EBAWA* [68] allows multiple instances of consensus to run in parallel (with different leaders) optimistically, further improving the performance of the system. Another strategy consists in carefully selecting the node depending on the network conditions to play the role of the leader. This strategy is used in *Archer* [24], where the system can perform a view change, not only when the current leader is faulty but also when another node could exhibit better performance than the current leader. Being careful in selecting the leader, or rotating the leader, can help the protocol but does not combat the inherent scalability issues. As at some point in the protocol, one or more processes have to broadcast a given value to all processes (bandwidth bottleneck) and receive and verify n signatures (CPU bottleneck).

3.2.1.1 Optimistic Protocols

To achieve consensus *PBFT* has to collect two subsequent quorums to ensure that if any correct node decides, all other correct nodes also decide until the end of the second round. Optimistic protocols are based on the observation that in fault-free runs, it may be possible to update the client about the result earlier. To achieve this, they run a combination of two sub-protocols: an optimistic sub-protocol that only provides termination in fault-free runs and a recovery sub-protocol that needs to run when faults occur (e.g., somehow inconsistencies are detected, or consensus is not achieved). The recovery protocol typically follows the structure of the original *PBFT* protocol, while the optimistic protocol typically uses fewer messages and/or fewer communication rounds. Thus, in failure-free runs, optimistic protocols can be more efficient but in faulty runs are more expensive (because both sub-protocols are executed sequentially).

WHEAT [64] is an example of a protocol that uses this approach. In *WHEAT*, the optimistic sub-protocol follows the same structure of *PBFT* but is executed only among a subset of $f + 1$ participants. Although the message complexity is still quadratic, namely $\mathcal{O}(n^2)$, in practical terms, it is more efficient than a protocol where at least $3f + 1$ nodes need to participate. If the optimistic protocol succeeds, the result is propagated to the remaining processes in the background; otherwise, a recovery protocol involving all nodes is executed. Another, even more optimistic approach has been implemented in *Zyzzyva* [38]. In *Zyzzyva*, nodes accept the value proposed by the leader without immediate validation: if the leader is faulty, participants may transiently store an inconsistent state. When inconsistencies arise, they need to be corrected by an expensive recovery protocol that uses a communication pattern similar to that of *PBFT*. In the fault-free runs, *Zyzzyva* allows clients to observe results after three communication steps that use only a linear number of messages. However, because participants may transiently store an inconsistent state, clients always have to contact a quorum of participants before accepting a value. If,

in this process, a client detects that servers are inconsistent, it will trigger the recovery protocol.

Ouroboros [33] is an extension of *Zyzzyva* where, instead of relying on the clients to detect inconsistencies, the results are published on the blockchain and are, if necessary, solved through forks. Thus, similarly to PoW blockchains, there is a possibility that blocks are revoked later on. Therefore, *Ouroboros* also does not offer deterministic finality.

While these optimistic protocols provide much lower latency and a significantly better message complexity in the failure-free case than the previously discussed protocols, in highly contentious settings, as we expect it in distributed ledgers, adversaries may regularly force the fallback protocol to be executed, resulting in a significant overhead and also a quadratic message complexity. On top of that, while these protocols offer a linear message complexity, a single process (leader) still has to broadcast the proposed value (batch of transactions) to all participants, which still creates a bottleneck.

3.2.2 Star Protocols

One of the main advantages of the *all-to-all* communication pattern adopted by *PBFT* and similar algorithms is that it does not require public-key cryptography. Instead, each point-to-point channel connecting a given pair of participants can be authenticated using MACs, which is computationally much more efficient. At the time *PBFT* had been designed, the use of public-key cryptography appeared to be more penalizing to the performance than the quadratic message cost of the algorithm. However, subsequent experience has shown that, with current technology, it may be better to use more sophisticated cryptographic techniques. This is the case, even though they might require more computational resources, as long as they allow to reduce the message and bandwidth complexity of the algorithm, given that processing a large number of messages also consumes a significant amount of resources [16, 19].

One way to reduce the communication cost of the algorithm is to establish relay points that aggregate a set of messages and relay it to a subset of participants. For instance, if one round of the protocol requires voting, instead of having participants send their votes directly to all other nodes, participants may send their votes to a selected participant (typically the leader), which returns the votes in aggregated form. This strategy allows to reduce the message complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. However, note that this scheme comes with a latency tradeoff: by using the leader as a relay point, voting now takes two communication steps instead of one.

In addition to that, this approach only works because public key cryptography prevents the leader from modifying the message contents. While the relay node can opt to drop some messages, it cannot change the content of the messages it chooses to relay. Due to this, several algorithms have been proposed that use a star topology to communicate.

The network communication pattern reduces the number of messages exchanged significantly but still results in a significant network resource consumption, as the relay node

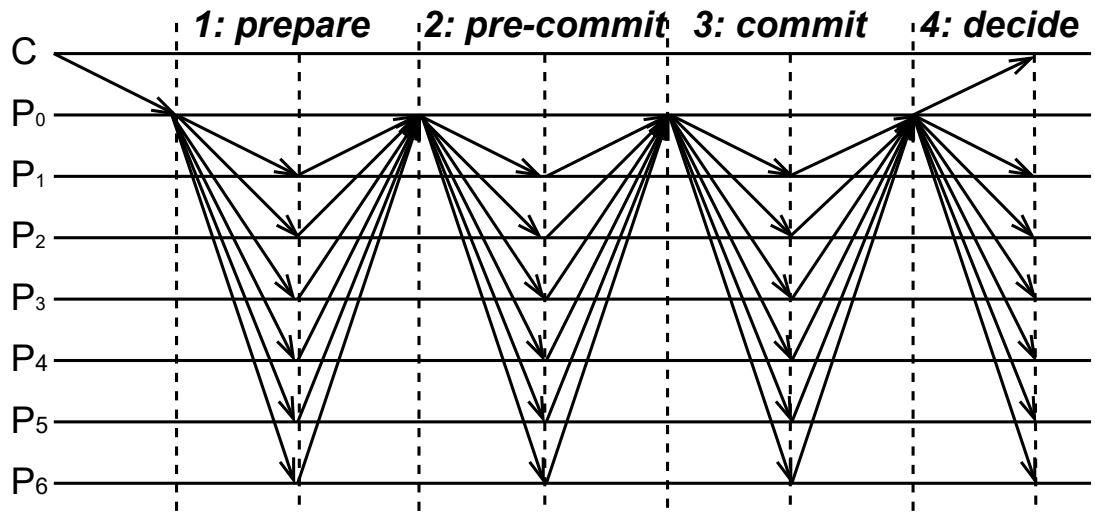

Figure 3.2: HotStuff: communication pattern in normal case operation.

still has to relay the content of all messages sent by all participants (or, at least, a quorum of participants). Signature aggregation may be used to construct one single signature that can be verified in $\mathcal{O}(1)$ steps. Using this technique, votes from the participants can be sent in the form of partial signatures that are then combined by the leader such that a single signature is sent back to all participants as proof that enough votes were collected.

HotStuff is a recent algorithm that uses a communication pattern based on a star-network [1]. *HotStuff* consists of three rounds where all communication runs through the leader that collects and distributes aggregated signatures. The protocol has a linear communication cost but takes eight communication steps to terminate (4 phases of two steps each). This process is shown in Figure 3.2.

The protocol is very similar to *PBFT*. In the first step, a client sends transactions to the current leader. The leader, similarly to PBFT, disseminates a proposal in a *prepare* message. Finally, if the proposal is valid (i.e., consists of valid transactions), processes send their vote to the leader as a response.

After aggregating $2f + 1$ votes the leader broadcasts the result in the *pre-commit* step. Processes verify the set of votes (i.e., the set of signatures or the signature aggregate) and, if valid again, signal the leader their agreement.

The leader again aggregates $2f + 1$ votes (this time over the *pre-commit*) and notifies the remaining processes in a *commit* message. In turn, after receiving a correct *commit* message, processes lock the result and again notify the leader.

In the final step, the leader aggregates $2f + 1$ *commit* response messages and broadcasts the decision. The remaining processes, after receiving a valid decision message (consisting of $2f + 1$ valid *commits*) then persist the result.

To compensate for the additional number of communication steps, *HotStuff* allows up

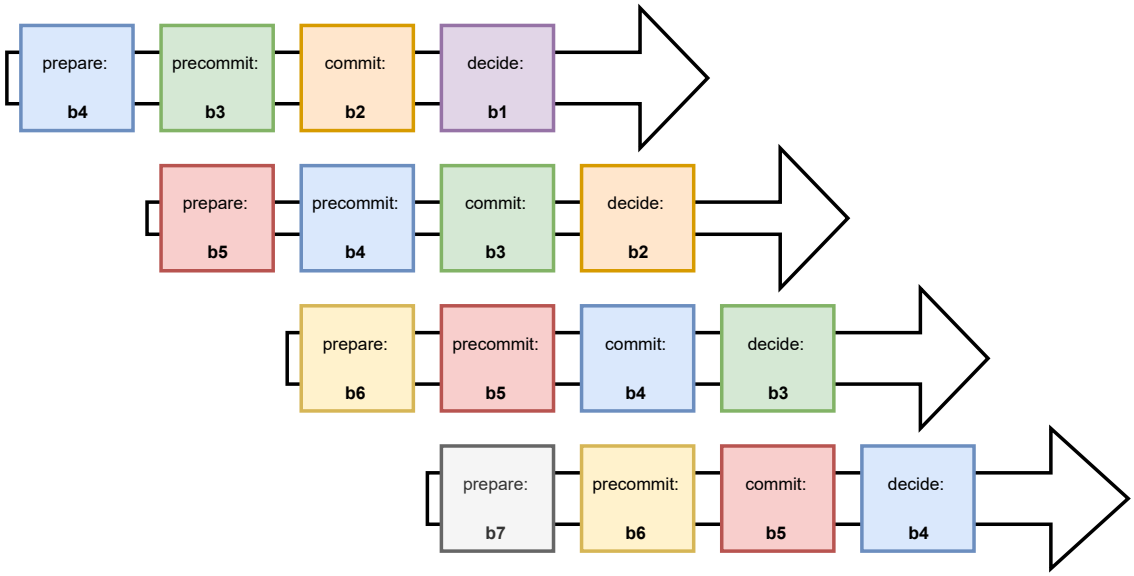

Figure 3.3: HotStuff: Pipelining

to four steps of different instances of consensus to run in parallel. This is displayed in Figure 3.3. More precisely, the first step of the $n+1^{th}$ instance of consensus is executed in parallel with the second step of the n^{th} instance of consensus, and so forth. It is important that the pipelining is done precisely in this manner, since in a blockchain scenario, the $n+1^{th}$ instance depends on the n^{th} instance.

Despite its apparent benefits in terms of message complexity, algorithms that use a communication pattern based on a star network have a significant drawback: the protocol becomes limited by the physical capacity of the leader node both in terms of bandwidth and CPU.

3.2.3 Gossip Protocols

Another possibility is using gossip message propagation. This approach is followed in *Gosig* [42], where the leader election is done by executing verifiable random functions. This approach decreases the risks of potential attacks on the elected leader as the leader is only known at the moment of the consensus and not ahead of time. While there is a chance of no or multiple leaders being elected in the same round, requiring a new election, this approach is still advantageous in highly adversarial environments. Protocols as *PBFT* or *Spinning* where the leader is always known beforehand can suffer denial of service attacks on their leaders, which can affect the liveness of the system. Additionally, the communication is optimized by relying on epidemic message propagation (gossiping) between the replicas and applying multi-signatures to improve the bandwidth usage. However, at the moment of writing, byzantine fault-tolerant gossip still requires at least multiple rounds of $\mathcal{O}(N * log(n))$ messages, which is more than $\mathcal{O}(\mathcal{N})$ of the star-based

approach.

3.2.4 Committee

As the communication and computation complexity is a function of the number of processes, instead of requiring all processes to participant in consensus, some approaches select a random subgroup of processes each round which then executes consensus. These approaches are usually called *committee*-based approaches, where SCP [7] and Algorand [26] are two of the most famous representatives.

However, this approach does not come without drawbacks. Due to the reduced quorum size, malicious processes may have a more significant influence on the system as there is a chance to encounter a committee with a majority of faulty nodes. As such, either the resilience has to be sacrificed (reducing the resilience to be in function of the committee size and not the system size) or deterministic finality where a block can only be finalized by sufficient subsequent blocks that guarantee that sufficient independent quorums validated a given block (implicitly vouching for previous blocks).

On top of that, at the end of each round, each committee member still has to broadcast the result to the remaining processes resulting in a high bandwidth cost for each of those processes.

3.2.5 Hierarchical Protocols

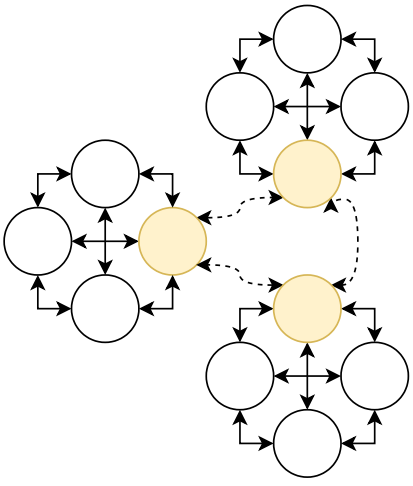

Figure 3.4: Hierarchy: Steward

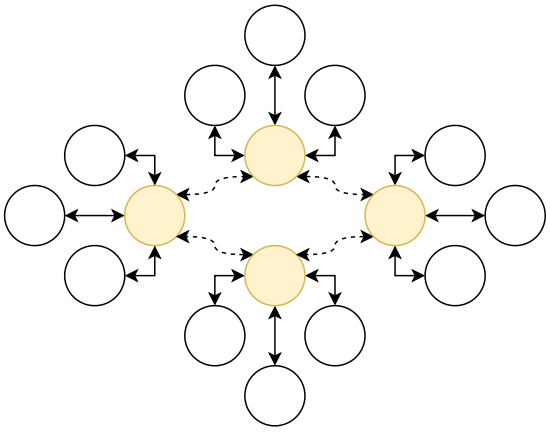

Figure 3.5: Hierarchy: Fireplug

A possible solution to divide the load among nodes are hierarchical architectures where nodes are split into groups (usually based on geographical proximity), execute consensus first on the group level, and finally, have a representative of each group participate in consensus on the global level.

Steward

One of the first to follow this approach is *Steward* [2]. An example hierarchy of the approach is shown in Figure 3.4 where servers are grouped in hierarchic groups (or clusters), and each cluster acts as a logical unit and individually creates a signature aggregate representing the decision of the group. As such, one process of each group may then use this signature aggregate when attempting to achieve consensus with the remaining group leaders on the "global level". This approach requires $\mathcal{O}(n^2)$ (where n represents the group size) messages within each group and $\mathcal{O}(k)$ messages on the global level (where k represents the number of groups). As a result, this very successfully reduces the communication complexity and distributes the load significantly. However, in comparison to *PBFT*, *Steward* offers significantly lower resilience guarantees, where, instead of having a global failure limit, failures are limited on a local level.

Fireplug

Fireplug [51], as depicted in Figure 3.5, reduces the cost of the architecture by requiring the leaders to attach a proof containing the signature of enough other leaders when sending decisions to their local cluster members. It runs a byzantine fault-tolerant protocol in the global group where a majority of leaders have to sign each decision before updating the client with the result. By deploying several local group representatives at the global level, it is possible to run the system with less than $3f + 1$ data centers overall. Nevertheless, while it may tolerate faults on any level of the hierarchy and local group members may detect a wide range of byzantine leadership behavior, a majority of byzantine leaders may slow down the system significantly. As such, the actual maximum number of simultaneous faults at the global level is restricted to $k = 3f + 1$ where k presents the number of replicas in the global group, which is unsuitable for highly adversarial environments since additional replicas in the local groups do not increase the worst-case resilience of the system.

ResilientDB

ResilientDB [58] is a more recent approach which combines hierarchical algorithms similar to *Steward* or *Fireplug* within a blockchain. In addition to the hierarchical approach, which distributes the load among the groups, *ResilientDB* leverages sharding to utilize the left-over resources to boost the system throughput significantly. Nonetheless, similar to the other hierarchical approaches, it comes with restrictive failure assumptions, as a majority of honest nodes is required within each sub-group.

3.2.6 Tree Based Protocols

As previous hierarchical protocols reduce the resilience to achieve higher throughput, approaches based on dissemination and aggregation trees attempt to distribute the load

equally among the internal nodes while still maintaining high resilience.

Byzcoin

One of the first approaches was Byzcoin [35]. Byzcoin leverages a binary tree and uses *collective co-signing* (a cryptographic scheme) to collect two quorums through a tree requiring three round-trips up and down the tree for each quorum. This approach, therefore, results in a significant latency overhead which affects the potential maximum throughput. In addition, Byzcoin uses verifiable random functions to construct a random tree each round, and if within a given time-frame no consensus is achieved, it falls back to an all-to-all scheme.

Motor

Motor [36] is a continuation of Byzcoin, which restricts the tree to a depth of two to decrease the actual latency cost, and uses BLS signatures that do not require several round-trips to construct a single quorum. It constructs trees in a similar manner as Byzcoin, but, instead of falling back to an all-to-all scheme, in the presence of failures, it slowly falls back to a star topology within $\frac{N}{m}$ steps (for a fanout of m). As such, in the worst case, it requires $f + 1 * \frac{N}{m}$ steps to find a robust configuration.

Omniledger

Omniledger [37] is the final evolution of Byzcoin and Motor and attempts to improve the throughput significantly through sharding by making each sub-tree a shard of the system that decides on a given subset of transactions and deploys a directed acyclic graph to deal with conflicting transactions. However, as each shard is smaller than the overall system size N, Omniledger trades resilience against additional throughput and, as it leverages acyclic graphs to achieve cross-shard consensus, it requires significantly longer to finalize a given block.

3.3 Discussion

Table 3.1 summarizes up the discussion about the different consensus protocols.

The first column specifies if an approach distributes the computational and bandwidth load among a set of processes. The second column defines if a system fulfills deterministic finality (i.e., a block is irreversible after a specific number of steps). Then, in the third column, we specify if a given approach displays optimal resilience (i.e., $N = 3f + 1$). Following that, a system fulfills the condition of the next column if it can provide high throughput independent of the network latency. Finally, we specify if an approach can recover deterministically in a linear number of configuration steps in the last column.

Table 3.1: Comparison of existing Algorithms

	Load balancing	**Deterministic finality**	$N = 3f + 1$ **resilience**	**Latency compensation**	**Quick recovery**
PBFT [16]	✗	✓	✓	✗	✓
HotStuff [1]	✗	✓	✓	✗	✓
Ebawa [68]	✗	✓	✓	✓	✓
Steward [2]	✓	✓	✗	✗	✗
Fireplug [51]	✓	✓	✗	✗	✗
ResilientDB [58]	✓	✓	✗	✗	✗
Multi-Layer [43]	✓	✓	✗	✗	✓
Algorand [26]	✓	✗	✓	✗	✗
SCP [7]	✓	✗	✓	✗	✓
Byzcoin [35]	✓	✓	✓	✗	✗
Omniledger [37]	✓	✓	✗	✗	✗
Kauri (This work)	✓	✓	✓	✓	✓

While traditional protocols like *PBFT*, *EBAWA*, and *HotStuff* provide deterministic finality, $N = 3f + 1$ resilience, and quick recovery, in either protocol, one or more processes have to broadcast the block and process and verify N signatures, which is a major bandwidth and CPU bottleneck, as such they do not offer any load-balancing properties. In addition, in a geo-replicated scenario, only *Ebawa* can compensate for the inherent latency overhead by running multiple consensus rounds and steps asynchronously in parallel. Nonetheless, *Ebawa* assumes non-conflicting values, which is not the case in the blockchain environment where each block extends the previous one.

Early hierarchical protocols offer the load balancing and deterministic finality the previous protocols lack, but at a tradeoff as they have to reduce their resilience significantly. In addition to that, none of these approaches achieves high throughput in a high latency setting, and only *Multi-Layer* offers a reconfiguration algorithm that allows quick recovery.

Modern committee based approaches like *SCP* and *Algorand* may reduce the cost significantly by having only a committee run the consensus. However, this is done at the cost of deterministic finality. In addition to that, neither approach deals with the inherent latency cost of a geographical deployment resulting in vast idle times.

Finally, existing tree-based approaches offer load balancing and deterministic finality but again lack compensation for geographical deployments. *Omniledger* attempts to compensate this partially through sharding but gives up a large part of their resilience in trade.

The system we have developed through the course of this thesis incorporates a series of novel techniques offering load balancing and latency compensation while maintaining deterministic finality, high resilience, and quick recovery in the majority of cases. How we achieve this is explained in detail in the next chapter.

3.4 Summary

This chapter introduced and discussed a series of consensus protocols that we classified by the communication structure. Furthermore, we highlight the disadvantages of the

different communication structures and the approaches that use them. Based on this, we introduce the system we have developed throughout this thesis and compare it to the existing approaches. The details of our system are outlined thoroughly in the following chapter.

4

Kauri

This chapter describes the proposed techniques and scientific contributions we have developed in the context of this P.h.D. After an initial discussion of the system model, we shed light on the tree communication scheme, how we achieve reconfiguration in optimal steps and how we leverage pipelining to achieve high throughput. Next, we discuss extended reconfiguration possibilities and close with Kauri, our prototype implementation.

The majority of the content of this chapter was peer-reviewed and published in our paper in [50]. Alongside the previously published content, we included more thorough explanations of the algorithms, an extended algorithm that allows tolerating more faults, and several alternative non-optimal tree construction approaches.

In the previous chapter, we identified a major bottleneck inherent to most BFT protocols. This bottleneck is twofold:

- Bandwidth: At least one process has to broadcast a value to all processes each round.
- CPU: At least one process has to verify all N signatures of each participant each round.

Due to this, with increasing numbers of processes, eventually, the system reaches its limit. While it is theoretically possible to scale the computational power of each process if sufficient resources are available, it is highly inefficient. The same does not apply for bandwidth, as in a geographically distributed environment, only a small share of the end-to-end bandwidth is under the influence of the server owner [58].

As such, we identified tree-based approaches as the best solution to distribute the load equally among a set of internal nodes. However, existing tree-based approaches still come with a set of drawbacks:

- Crippled throughput due to a large number of communication steps.
- Vulnerability of tree structures in the presence of byzantine faults.

First, due to the added number of communication steps, in a geographically distributed environment, processes will spend a large percentage of the time idling while waiting for the communication to complete.

Second, while in an all-to-all scheme, a faulty process may at most withhold their own vote, in a tree-based communication scheme, a faulty internal node may easily prevent consensus from being reached.

To solve the above problems, we came up with two solutions, namely optimistic pipelining to fully leverage the idle time and a reconfiguration algorithm that allows constructing trees without faulty internal nodes in optimal time.

How exactly this is achieved is discussed in the following sections.

4.1 System Model

We assume the existence of N server processes $p_0, p_1, p_2, ..., p_{N-1}$ and a set of S client processes $c_0, c_1, c_2, ..., c_{S-1}$. Client and server processes are connected through perfect point-to-point communication channels constructed by adding mechanisms for message retransmission as well as detecting and suppressing duplicates [15]. With the help of these mechanisms, the channels fulfill the following properties:

- *Validity:* If a process p_j receives a message ms over a channel e_{ij}, ms was sent by p_i.
- *Termination:* Given correct processes p_i and p_j , if p_i sends ms over the channel e_{ij} connecting both processes p_j eventually receives ms.

We assume the Byzantine fault model, where Byzantine processes may produce and return arbitrary values, delay and omit messages, and collude with each other. However, they do not possess sufficient computational power to compromise cryptographic primitives (e.g., forging signatures or finding colliding hashes). Based on this premise, a correct process is a process that follows its specification; else, it is considered faulty. Of the total set of N processes, at most $f \leq \frac{N-1}{3}$ can be faulty. Furthermore, to respect the FLP condition [25] we assume a partially synchronous system where safety is always guaranteed, but progress is only made during synchronous periods [23] (see Section 3).

As we assume a distributed ledger environment, processes use their public keys as unique identifiers. Each message ms of a given process p_i is signed using their private key $priv_i$. Thus, any other process p_j may identify the origin of a signed message ms_i and verify the message integrity (the message was not tampered with) with the help of the public key pub_i.

4.2 Tree Communication

In this section, we discuss tree communication and show how it fulfills safety and liveness in the context of byzantine fault-tolerant consensus. Interestingly enough, the star used in HotStuff [1] is actually a particular case of a tree of depth 2 with a single internal node (We explained HotStuff in detail in Section 3). Due to this observation, instead of creating a new consensus protocol from scratch, we solely have to adapt certain primitives used in

HotStuff to support tree-based communication and prove that they do not alter the safety and liveness guarantees inherent to HotStuff.

4.2.1 Communication in HotStuff

HotStuffs communication can be summarized as two phases. First, the broadcast of the leader process, where the leader process disseminates data to all processes, and, second, the aggregation step, where the leader awaits $N-f$ votes from all processes. Each phase is executed once each round for a total of three rounds to finish a given consensus instance and collect three consecutive quorums.

On the basis of this we define the following two primitives:

- *broadcast(data).* *data* is broadcast by the leader to all processes.
- *await (N - f) votes.* Leader awaits $N-f$ votes as responses to the previous broadcast.

However, in certain situations, these primitives might fail. First of all, during asynchronous system conditions, it is possible that the broadcast of the leader does not reach a subset of processes. Thus, the leader could be awaiting votes indefinitely. Second, a faulty leader could only disseminate messages to a subset of processes.

Based on this observation, we define the following property:

Definition 1 ***Strongly Robust Star:*** *A star is said to be strongly robust if the leader is correct and non-robust if the leader is faulty.*

Given the robustness criteria, in HotStuff, the primitives have to fulfill two additional properties:

Definition 2 ***Reliable Dissemination:*** *After GST, in a strongly robust configuration, a quorum of correct processes receives the data sent by the leader.*

Definition 3 ***Byzantine Quorum:*** *After GST, in a strongly robust configuration, the leader is able to collect at least $N-f$ votes.*

Assuming a strongly robust star and perfect point-to-point channels, Reliable Dissemination and Byzantine Quorum can be fulfilled fairly trivially.

Briefly, due to the perfect point-to-point channels and the assumption that the leader is correct, it is guaranteed that all processes eventually receive the *data* disseminated by the leader. As such, Reliable Dissemination is fulfilled. Furthermore, due to the perfect point-to-point channels and the fact that no more than f processes are faulty, the leader is guaranteed to receive at least $N-f$ votes eventually. As such, the Byzantine Quorum is also fulfilled.

Given the fulfillment of the above properties, the protocol proposed in HotStuff can achieve both safety and liveness. For the complete proofs, we refer to the HotStuff paper [1].

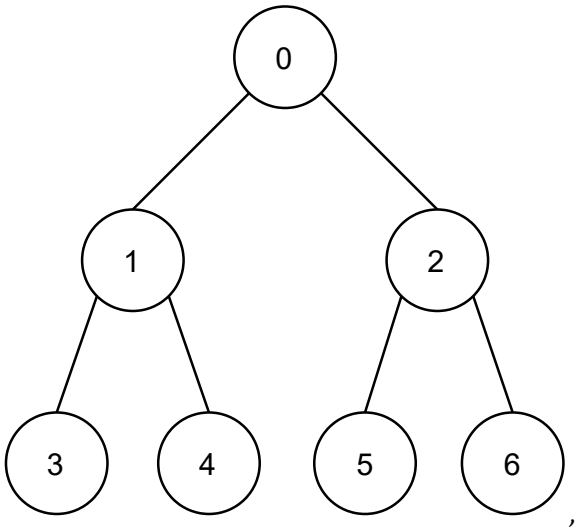

Figure 4.1: Simple Tree Example

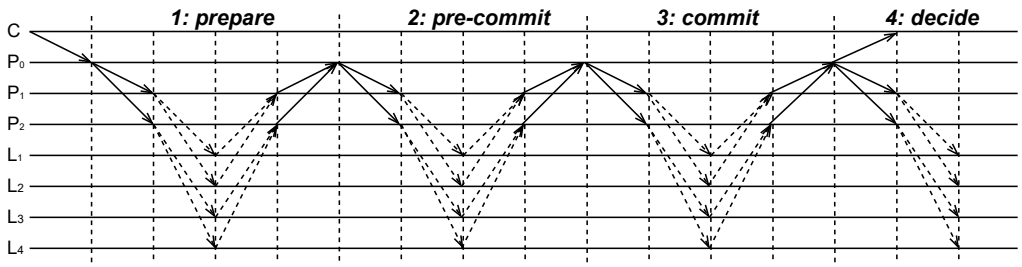

Figure 4.2: Tree communication pattern for 7 processes.

4.2.2 Communication in a Tree

Next, we discuss the adaption of the BROADCAST and AWAIT primitives to tree-based communication while making sure that we are still able to observe the same properties.

We organize processes in a tree structure, with the leader process at the root. Instead of having the leader BROADCAST the *data* to all processes, the leader disseminates *data* to their child nodes. The child nodes, in turn, forward it to their own children until reaching the leaf nodes. Adjacently, the primitive *await* has leaf nodes disseminate their vote to their respective parent nodes, which in turn aggregate the data of their leaves and further hand it to their parents until reaching the root. We illustrate this process in Figure 4.2 for the example tree of Figure 4.1.

As in this algorithm, not only a faulty leader may influence the outcome, but also faulty intermediary nodes (internal nodes) may hinder progress; not only do we have to adjust the notion of a robust configuration in the context of tree-based dissemination, but there are actually two possible robustness definitions.

Definition 4 ***Strongly Robust Tree:*** *A tree is strongly robust iff the leader process is correct, and there is a path of correct processes connecting any given correct process to the root. .*

Algorithm 1 Impatient Channels: RECEIVE

```
1: function IC.RECEIVE(p)          ▷ where IC is an impatient channel built on top of perfect channel PC
2:     on PC.RECEIVE (p, ms) return ms
3:     on TIMEOUT(Δ) return ⊥
4: end function
```

While strong robustness guarantees consensus for $f = \frac{N-1}{3}$ and any distribution of failures, in many cases, this definition of robustness is unnecessarily strong. In fact, as long as there is a path between the leader and a quorum of correct processes, Reliable Dissemination is fulfilled, and consensus may be reached.

Definition 5 ***Robust Tree:*** *A tree is robust iff the leader process is correct and there is a path of correct processes connecting a quorum of correct processes to the root. .*

However, even a strongly robust tree does not necessarily guarantee Reliable Dissemination and Byzantine Quorum compared to a star. For example, a faulty internal process might prevent a correct child process from sending its vote. Another example might be a faulty leaf node that prevents a correct internal node from progressing by never sending their vote.

4.2.2.1 Impatient Channels

We solve this with the help of impatient channels. Perfect point-to-point channels only guarantee the delivery of a message if both processes that are communicating are correct. Hence, a faulty sender could simply omit to send a message and block the receiver indefinitely. However, we have to guarantee that any process eventually makes progress. We guarantee this through the usage of impatient channels. Impatient channels always return a value, either the value disseminated by the sender or $\perp$ after a timeout.

However, after GST, if both participating processes are correct, the receiver is guaranteed to receive the sent value. Impatient channels a SEND and RECEIVE primitive with the following properties:

- *Validity:* If a process p_j receives a message ms over a channel e_{ij}, ms was sent by p_i or $ms = \perp$
- *Termination:* RECEIVE always eventually returns some value or $\perp$.
- *Conditional Accuracy:* Assuming both channel participants p_i and p_j are correct, after GST, p_j always returns ms sent by p_i.

We show an example implementation of the RECEIVE method of impatient channels in Algorithm 1 assuming the known bound Δ as the worst-case network latency.

Thus, with the help of impatient channels, we can mitigate the impact faulty leaves might have on their parent node.

4.2.2.2 Cryptographic Collections

Similar to a star topology, in a tree topology, we require cryptographic signatures to guarantee the integrity of messages when being passed through the tree. However, if the internal nodes only relay a set of signatures, the process at the tree's root has to verify all N signatures, leading to a bottleneck and impairs the protocol's scalability.

As such, we leverage a cryptographic aggregation scheme to mitigate these costs. This way, votes are aggregated on the way to the root, and both the internal nodes and the root have to verify at most m messages, where m is the maximum fanout of the tree.

To simplify the presentation we model the vote aggregation scheme with the help of an abstraction we call a *cryptographic collection*. This collection corresponds to a set of (p_i, ms_i), where p_i identifies a given process and ms_i their signature. A new collection c is created by a given process p_i with a signed message ms_i by calling c=NEW$((p_i, ms_i))$.

Furthermore, the primitive $c_{12} = c_1 \oplus c_2$ describes a combination of two given collections. One of the most important properties of this collection is that it allows any given process to verify if the given collection c reached a certain threshold t of signatures regarding the same message ms. This is done through the HAS(c, v, t) primitive. In order to verify the total size of said collection c, we check its cardinality $|c|$.

As a result, our cryptographic collections fulfills the following properties:

- *Commutativity:* $c_1 \oplus c_2 = c_2 \oplus c_1$
- *Associativity:* $c_1 \oplus (c_2 \oplus c_3) = (c_1 \oplus c_2) \oplus c_3$
- *Idempotency:* $c_1 \oplus c_1 = c_1$
- *Integrity:* Let $c = c_1 \oplus \ldots c_i \ldots c_n$. If HAS (c, v, t) then at least t distinct processes p_i have executed c_i=NEW$((p_i, v))$

In the context of this work we leverage non-interactive BLS signatures. BLS signatures are a multi-signature scheme that allows to aggregate a set of signatures to a single signature with a constant size that can be verified in $\mathcal{O}(1)$ steps. As such, each internal node in the tree may aggregate all their votes into a single aggregated signature that is then sent to their parent, where it may be verified in $\mathcal{O}(1)$ steps. The resulting overall cost is $\mathcal{O}(m)$ at each internal node, including the root, where m is the fanout of the tree. Note that classical asymmetric signatures require $\mathcal{O}(N)$ verifications at each process [11].

4.2.2.3 Implementing *broadcast* and *await*

Now that we established the primitives that are required for message passing in a tree structure, we look at how we leverage these primitives to implement *broadcast* and *await*.

Algorithm 2 presents the pseudocode for *broadcast* that is executed on all processes in the system. When distributing the *data* in one of the phases, a process expects to RECEIVE *data* from their parent which is then relayed to their child processes. Due to the

Algorithm 2 *broadcast* on a tree G (process p_i)

```
1: procedure BROADCAST(G, data)
2:     children ← G.CHILDREN(p_i)          ▷ Get edges to children of p_i
3:     parent ← G.PARENT(p_i)              ▷ Get parent of p_i (returns ⊥ for root)
4:     if parent ≠ ⊥ then
5:         data ← IC.RECEIVE(parent)       ▷ Receive from parent
6:     end if
7:     if data ≠ ⊥ then
8:         for all e ∈ children do         ▷ Send to children
9:             IC.SEND(e, data)
10:        end for
11:    end if
12:    return data
13: end procedure
```

impatient channel, this process always terminates even in the presence of faulty internal nodes (after reaching timeout Δ).

Theorem 1 *After GST, Algorithm 2 fulfills Reliable Dissemination in a strongly robust tree.*

Proof 1 *We prove this by contradiction. Assume Reliable Dissemination is not fulfilled. As such, one or more processes must not have received the data that the leader sent. While Reliable Dissemination only requires a quorum of correct processes, in the worst case, in the presence of $f = \frac{N-1}{3}$ failures, all correct processes are required to participate. This implies that one of the following must be true: i) At least one correct process is either not directly connected to the leader or not connected to the leader through a path of correct processes. ii) The data got lost in the channel. iii) A correct process did not follow its specification.*

First, trivially, the definition of a correct process requires it to follow its specification. Similarly, as our communication is based on perfect point-to-point channels (further extended by impatient channels), following the termination property, messages always reach the recipient.

As such, only the first option is left over. However, we assume a robust configuration, which, by definition, ensures a correct leader and a path of correct processes from sufficient correct process to the leader. This leads to a contradiction. Thus, Algorithm 2 guarantees Reliable Dissemination.

Now that we have proven the fulfillment of Reliable Dissemination, we show that we can also fulfill the Byzantine Quorum requirement. The implementation of *await* that achieves this is displayed in Algorithm 3.

The implementation of *await* leverages cryptographic primitives to efficiently aggregate signatures on the way to the root process. Note that this is not a requirement to collect a Byzantine Quorum. Similarly to *broadcast*, due to the use of impatient channels, *await* is also guaranteed to terminate. This is particularly important for *await*, as otherwise faulty leaf nodes might stall their parent indefinitely and prevent their parent process from relaying the signatures they received from the remaining correct processes.

The implementation of *await* does the inverse of the *broadcast*. As such, each process awaits votes of their child processes, aggregates them, and relays them to their parent process.

Theorem 2 *After GST, Algorithm 3 guarantees a Byzantine Quorum in a strongly robust tree.*

Algorithm 3 *await* on a tree G (process p_i)

```
1: procedure AWAIT(G, input)
2:     children ← G.CHILDREN(p_i)                ▷ Get edges to children of p_i
3:     parent ← G.PARENT(p_i)                    ▷ Get parent of p_i (returns ⊥ for root)
4:     collection ← NEW((p_i, input))
5:     for all e ∈ children do                   ▷ Empty for leaf nodes
6:         partial ← IC.RECEIVE(e) it
7:         collection ← collection ⊕ partial
8:     end for
9:     if parent ≠ ⊥ then
10:        IC.SEND(parent, collection)
11:    end if
12:    return collection
13: end procedure
```

Proof 2 *We again prove this by contradiction. As such, we have to assume that the leader could not collect a quorum of signatures. Thus, based on Algorithm 3 this implies that either: i) An internal node did not receive the signatures from all its correct children (line 6). ii) An internal node did not aggregate and relay all signatures it received from its correct children (line 10). iii) An internal node got stuck waiting for all signatures.*

First, due to the perfect point-to-point channels under the premise that correct processes follow their specification; it is not possible that a vote of a correct child process does not reach its parent. Similarly to the first case, a correct internal node always follows its specification and relays the aggregated votes. Finally, a correct node might get temporarily stuck waiting for additional votes. However, due to the implementation of impatient channels, eventually, the channel returns $\perp$, unblocking the internal node and allowing the internal node to relay the remaining votes they have received until Δ.

Thus, any of the options leads to a contradiction and, therefore, Algorithm 3 guarantees a Byzantine Quorum.

As such, with the help of the implementation of the two primitives (*broadcast* and *await*), it is possible to replace the star topology with a tree topology while still offering the same guarantees (Reliable Dissemination and Byzantine Quorum).

Thus, a tree structure fulfills the same safety guarantees as HotStuff [1], as replacing the underlying communication structure does not alter the protocol (three subsequent quorums, block validity, and conflict resolution). While this is less obvious in the case of liveness, due to the implementation of impatient channels, faulty nodes in a tree do not have more power than in HotStuff (i.e., a faulty internal node is at most as powerful as a faulty leader, and faulty leaf nodes may delay the system at most by some Δ).

However, to achieve liveness, we require a robust communication structure which is significantly more complex to achieve for a tree compared to a star. In addition to that, due to the increased number of communication steps, the throughput of a tree topology is more sensitive to the underlying system latency.

Therefore, in the following two sections, we discuss how we deal with reconfigurations and how we compensate latency to achieve high throughput in high latency environments.

4.3 Reconfiguration

In the previous section, we have discussed the necessary requirements of robust graphs for consensus to terminate. We now discuss how to build reconfiguration strategies for different topologies.

First, we present some preliminary definitions required to construct robust trees. Next, to illustrate our approach, we first define an evolving star in Section 4.3.2 which is similar to the widely used rotating leader approach. Next, in Section 4.3.3 we present an algorithm for building evolving trees.

4.3.1 Preliminaries

As we consider a partially synchronous system, reconfigurations might not only be triggered by faulty processes but also by asynchronous system conditions. As such, a reconfiguration algorithm ought to present certain characteristics to operate accordingly under these conditions.

We model a sequence of configurations (static graphs) with the help of an evolving graph that fulfills the following property:

Definition 6 ***Recurringly Robust Evolving Graph:*** *A given evolving graph $\mathcal{G}$ observes recurringly robustness iff in an infinite sequence of reconfigurations robust configurations also appear infinitely often.*

While this is sufficient to guarantee that we eventually reach a robust graph that allows consensus to terminate, in practice, a large number of reconfiguration steps is undesirable. Not only does the system have to reconfigure successfully within a given period of synchrony, but the system is also essentially halted during the reconfiguration process and, as such, is unable to deliver its service.

Ideally, we want to find a robust configuration after a relatively small number of reconfigurations t. We call this property of evolving graphs *t-Bounded conformity*.

Definition 7 ***t-Bounded Conformity****: A recurringly robust evolving graph $\mathcal{G}$ observes t-Bounded Conformity if a robust configuration is reached every t reconfiguration steps.*

In practice, modern protocols like *PBFT* or *HotStuff* are able to reconfigure in $f+1$ steps which is optimal for leader driven protocols. This property arises from the fact that given a protocol that depends on the correctness of a specific process (e.g., the leader), in the presence of f faulty processes, in the worst case, we might elect f subsequent faulty processes for this specific role. As such, only after $f+1$ attempts, we reach a correct process, and consensus can be achieved. We call this property: Optimal Conformity.

Definition 8 ***Optimal Conformity****: A recurringly robust evolving graph $\mathcal{G}$ observes Optimal Conformity if a robust configuration is reached every $f+1$ reconfiguration steps.*

For many topologies achieving this is challenging. Hence previous works fall back to an all-to-all or star communication pattern when consensus cannot be reached using an alternative topology [35, 37][1]. We are interested in defining evolving graphs that avoid falling back to a different topology. Namely, we consider evolving graphs built exclusively of the same star or tree topologies. This way, we aim to preserve the topologies' appealing scalability and load-balancing properties, rather than falling back to a degraded state upon failures.

4.3.2 Strongly Robust Stars

An evolving graph based on a star topology can be constructed using the BUILD primitive shown in Algorithm 4. The evolving graph is constructed by letting each static graph G_k be a star whose center is given by process $p_{(k \bmod N)}$. This is equivalent to the rotating leader strategy that is used in a large number of leader-based consensus protocols. The primitive returns a tuple consisting of the root, outbound graph G, and inbound graph G^T.

Algorithm 4 Construction of an Evolving Star

```
1:  procedure BUILD(k)
2:      G_k ← ∅
3:      V_k ← N                                   ▷ set of vertices
4:      root_k ← p_(k mod |N|)
5:      leaves ← V_k \ {root}                     ▷ Remaining processes are leaf nodes
6:      for all v ∈ leaves do                     ▷ There is an edge connecting the center with each leaf
7:          G_k ← G_k ∪ {(root, v)}
8:      end for
9:      return(root, G_k, G_k^T)
10: end procedure
```

Theorem 3 *An evolving graph $\mathcal{G} = \{G_1, \ldots, G_k, \ldots\}$ where G_k is defined by the function BUILD(k) depicted in Listing 4 satisfies Optimal Conformity.*

Proof 3 *Algorithm 4 builds a total of N distinct star graphs where the center of the star is chosen deterministically as a function of k (line 4) and the remaining processes are leaf nodes. Assuming at most f faulty processes where $N = 3f + 1$, there are at most f sequential graphs with a faulty process at the center. Hence there are at most f sequential non-robust graphs. Therefore, for any $f + 1^{th}$ consecutive graph, there is at least one graph G_i that has a correct process at the center. Hence in G_i, there is a path composed exclusively of safe edges between every correct process, thus fulfilling the Strong Robustness property. Therefore, the BUILD function of Algorithm 4 defines an evolving graph that satisfies Optimal Conformity.*

It is important to note that this only holds under synchronous system conditions. It is not possible to build a robust graph in any topology during asynchronous phases.

[1] Note that these approaches thus require at least $x + f + 1$ reconfigurations in the worst case since x reconfiguration changes the topology to a star or clique and then $f + 1$ additional leader changes are necessary in the worst case.

However, as already mentioned, if there is a finite set of reconfiguration steps until a correct configuration is found, eventually, after GST, a robust configuration will be found in at most $f+1$ steps. For the sake of simplification, we omit this discussion in the proofs regarding the remaining reconfiguration algorithms as all proposed algorithms terminate in finite steps. In this case, as there are N processes that might be eligible as a leader, there is also a finite number of different configurations.

4.3.3 Strongly Robust Trees

We now discuss how to reconfigure strongly robust trees. Note that this is significantly harder than in the case of stars due to the much larger number of possible configurations. In fact, while in a star topology, given N processes, there are at most N distinct graphs. In the case of binary trees, for example, the number of possible graphs is given by the Catalan number $C_n = \frac{(2N)!}{((N+1)!N!}$. Additionally, there are $N!$ possible assignments from processes to nodes in the graph for each tree. To make the problem tractable, we restrict the algorithm to evolving graphs with a single topology where only the assignment of processes to nodes in the graph is altered. Nonetheless, the number of possible configurations is still factorial to the number of processes.

For simplification reasons, for now, we assume the Strong Robustness criteria. As such, we attempt to build a tree without any faulty internal nodes.

Following combinatorics, for a graph with N processes and I internal nodes including the root, there are $\binom{N}{I}$ (also denoted as $\frac{N!}{I!(N-I)!}$) different assignment possibilities.

However, due to the presence of $f = \lfloor \frac{N-1}{3} \rfloor$ faulty processes, not all these combinations are robust. Assuming f faulty processes, there are $\frac{(N-f)!}{I!(N-f-I)!}$ robust graphs. If we divide this by the total number of graphs $\frac{N!}{I!(N-I)!}$ we get the probability p_I to build a robust static graph G_k (Displayed in Equation 4.1)[2].

$$p_I = \frac{\frac{(N-f)!}{I!(N-f-I)!}}{\frac{N!}{I!(N-I)!}} \tag{4.1}$$

For a star topology where $I = 1$, solving this is trivial since $\frac{N!}{1!(N-1)!} = N$ and based on that, $\frac{(N-f)!}{1!(N-f-1)!}$ results in the probability $p_I = \frac{N-f}{N}$. Thus, for $N = 3f+1$, the probability of constructing a robust static graph G_k by sequential shuffling stays constant for any size of N. Therefore, a robust configuration is guaranteed within optimal steps even when using a naive reconfiguration strategy.

However, for tree topologies, where the number of internal nodes I usually increases with the total number of processes N, the probability p_I to construct a robust graph is

[2]This equation could also be displayed in the form of a hypergeometric distribution. We omit this as we aim to continue processing this equation in the next step.

Algorithm 5 Construction of a Strongly Robust Tree

```
1: function INIT(𝒩, m)                          ▷ Initialize the evolving tree with the set of nodes 𝒩 and fanout m
2:     B ← ∅                                    ▷ Initialize the set of bins
3:     for all i ∈ 𝒩 do                         ▷ For each Node
4:         B^{i mod m} ← B^{i mod m} ∪ i        ▷ Assign the node i to one of the m bins
5:     end for
6: end function
7: function BUILD(k)
8:     i ← k mod m
9:     𝒢^i ← all possible trees whose internal nodes are drawn exclusively from B^i.
10:    T^k ← pick any tree at random from 𝒢^i
11:    return T^k
12: end function
```

much lower. Since f directly depends on N, we may replace f in Equation 4.1 with $\lfloor \frac{N-1}{3} \rfloor$ resulting in Equation 4.2.

$$[h]p_I = \frac{\frac{(N-\lfloor \frac{N-1}{3} \rfloor)!}{I!(N-\lfloor \frac{(N-1)}{3} \rfloor - I)!}}{\frac{N!}{I!(N-I)!}} \tag{4.2}$$

Based on this, the main factor determining the probability p_I is the number of internal nodes I. For example, considering $\lim_{N\to\infty} p_I$ for $I = 4$ the probability to build a robust graph, in the average case, results in around 20% and for $I = 10$ already only at around 1.7%. As the total number of internal nodes I in the tree increases, the probability of finding a robust tree using a naive or randomized strategy becomes vanishingly small. In fact, in the worst case, the required number of configurations is factorial to the number of internal nodes I.

4.3.3.1 Perfect m-ary Trees

We now show that it is possible to construct an evolving graph $\mathcal{G}$ consisting exclusively of perfectly balanced m-ary trees that satisfy Optimal Conformity, as long as $f < m$. Let us assume a perfect m-ary tree, i.e., all internal nodes have m children, and all leaves have the same depth or level. In this case, for any m-ary tree, the number of internal nodes is always smaller than $\frac{N}{m}$ [46]. Our construction is based on the observation that a tree for which no internal node is faulty is trivially robust. Therefore, we reduce the problem of finding a robust tree to finding a set of correct internal nodes. To achieve this, we start by splitting the set of N processes in m disjoint bins B^i, each containing at least $\frac{N}{m}$ processes, i.e., $\mathcal{N} = B^0 \cup B^1 \cup \ldots \cup B^{m-1}$. As long as the actual number of failures f_a conforms to $f_a < m$, at least one of these bins contains no faulty processes. Let $\mathcal{G}^b$ include all possible m-ary trees whose internal nodes are drawn exclusively from bin B^b. Note that, under the assumption of a perfect m-ary tree, it is always possible to build such a tree where the internal nodes are drawn from bin B^b and the leaf nodes are drawn from the remaining bins. As at least one bin B^i contains solely correct processes, any tree whose internal nodes are drawn exclusively from bin B^i is guaranteed to be robust. To ensure Optimal Conformity it is therefore enough that the evolving graph G selects, for iteration k, one of the m-ary trees $\mathcal{G}^{k \bmod m}$. This algorithm is illustrated in Algorithm 5.

Theorem 4 *An evolving graph $\mathcal{G} = \{G_1, \ldots, G_k, \ldots\}$ where G_k is defined by the function* BUILD*(k) depicted in Listing 5 satisfies Optimal Conformity.*

Proof 4 *Processes are split among m disjoint bins. Since, by assumption $f_a < m$, it is guaranteed that at least one of the m bins solely consists of correct processes allowing to construct a static graph G_k with solely correct processes as internal nodes. Moreover, because we deterministically and sequentially iterate over the bins from which the internal nodes are drawn (line 8), it is guaranteed that at most after $f_a + 1$ iterations, we will select a bin with only correct processes. A tree containing correct internal nodes is trivially robust. Thus, the above construction builds a robust tree after at most $f_a + 1$ iterations. Hence ensures that the resulting evolving graph satisfies Optimal Conformity.*

4.3.3.2 Generic Trees

While we have shown that it is possible to achieve Optimal Conformity with perfect m-ary trees, the construction so far only works for perfectly balanced $m - ary$ trees.

Generally speaking, the only way to achieve Optimal Conformity is to split all processes into $f + 1$ bins (similarly to the m-ary trees). Such that, there is at least one bin without any faulty processes from which internal nodes are drawn. Thus, considering N total processes and I internal nodes, $\frac{N}{I} \geq f + 1$ has to hold (i.e. there are sufficient processes to divide all processes into $f + 1$ bins of size I). We prove this next.

Theorem 5 *An evolving graph $\mathcal{G}$ of generic trees satisfies Optimal Conformity iff $\frac{N}{I} \geq f + 1$, where I is the number of internal nodes for $f = \frac{N-1}{3}$.*

Proof 5 *We prove this in two steps. First, we prove that if $\frac{N}{I} \geq f + 1$ Optimal Conformity can be achieved, then, in the second step, we prove that Optimal Conformity in a generic tree can only be achieved if and only if $\frac{N}{I} \geq f + 1$.*

Similar to Theorem 4, we divide all processes into bins, and in each iteration k, the processes of a distinct bin are selected as internal nodes.

Trivially, to divide all processes into bins of size I, we divide the total number of processes N by the number of internal nodes I resulting in $\frac{N}{I}$ bins. Thus, as long as $\frac{N}{I} \geq f + 1$ there are $f + 1$ or more bins and, following Theorem 4, Optimal Conformity is observed.

In the second step, we prove that, in the worst case, it is not possible to achieve Optimal Conformity deterministically unless this condition is met. We prove this by contradiction. Thus, we assume there has to be a tree topology that can achieve Optimal Conformity where $\frac{N}{I} \leq f$.

As discussed, in the presence of $f = \frac{N-1}{3}$ failures, a set of solely correct internal nodes is required to guarantee the construction of a robust tree. However, in the worst case, it is impossible to detect faulty internal nodes. As such, we require disjoint sets of internal nodes. Thus, in order to achieve Optimal Conformity, we require a sequence of $f + 1$ disjoint sets of internal nodes such that, in the worst case, the $f + 1^{th}$ set consists solely of correct processes.

Considering $\frac{N}{I} < f$, it is not possible to divide all processes in $f + 1$ bins of the size of I with distinct processes in each bin. Thus, to create $f + 1$ bins, at least one node must appear in

multiple bins. However, this could be a faulty node, resulting in a faulty node in each of the $f+1$ bins. Similarly, if the processes are divided into less or equal to f bins, each bin may also contain at least one faulty node.

As such, either the tree structure has to tolerate faulty internal nodes, or it has to be possible to detect a faulty internal node to exchange it. Otherwise, more than $f+1$ steps are necessary to find a robust configuration. The first is not possible as there are no redundant communication paths in a tree and, in the worst case, in the presence of $f = \frac{N-1}{3}$ failures, a faulty internal node can block a correct node from participating. The latter is also not possible as it is impossible to differentiate between a faulty leaf that did not send a message in time, and a faulty internal node that disregarded a message of a correct leaf node. Therefore, in the presence of $f = \frac{N-1}{3}$ more than $f+1$ steps are necessary to find a robust configuration for $\frac{N}{I} \leq f$ leading to a contradiction as at most $f+1$ steps are allowed to observe Optimal Conformity.

As proven, $\frac{N}{I} \geq f+1$ presents the upper limit to achieve Optimal Conformity for any generic tree for $f = \frac{N-1}{3}$. Considering $f = \frac{N-1}{3}$ faults, $\frac{N}{I} = f+1$ results in $\frac{3f+1}{I} = f+1$ which can be simplified to $\frac{3f+1}{f+1} = I$ which for $\lim_{f \to \infty}$ diverges at 3. As such, while for $f = \frac{N-1}{3}$ we can achieve Optimal Conformity for generic trees, in practise the approach limits the type of tree significantly and allows to distribute the load between at most two processes equally.

4.3.3.3 Linear Conformity

While it is not possible to construct a tree in optimal steps if $\frac{N}{I} < f+1$ for $f = \frac{N-1}{3}$ (as proven in the previous section) there are still ways to construct robust trees in $\mathcal{O}(N)$ steps.

In the following, we describe an algorithm that achieves very close to Optimal Conformity using clever combinatorics.

When dividing the total number of processes N by the number of internal nodes I, there is a possibility that the result is a fraction between f and $f+1$. Thus, while it might not be possible to divide all processes into $f+1$ distinct bins, it is possible to divide all processes into f distinct bins and have L leftover processes that are not assigned to any bin. If in the worst case, none of the f bins results in a robust tree, we can safely assume that there is one faulty process in each bin and that none of the remaining processes in L may be faulty. Thus, in the next step, we may replace the I processes in one of the f bins, step by step, with the leftover processes of L until a robust tree is found. This way, we can construct a robust tree in $f + \frac{I}{L}$ steps in the worst case (since none of the L leftover processes is faulty, with multiple leftover processes, we can replace multiple processes in the bin at a time).

In Figure 4.3 we show an example of the outlined algorithm. Assuming a balanced binary tree of depth 2 resulting in a total of $N = 7$ processes and $I = 3$ internal nodes and, following $N = 3f+1$ there are $f = 2$ faulty nodes (highlighted red in the figure).

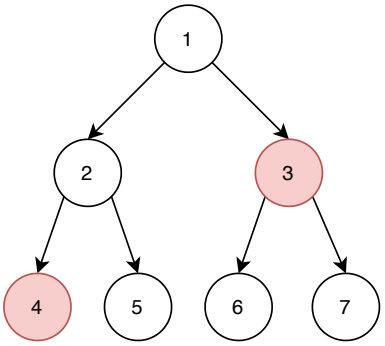

Figure 4.3: Simple Binary Tree

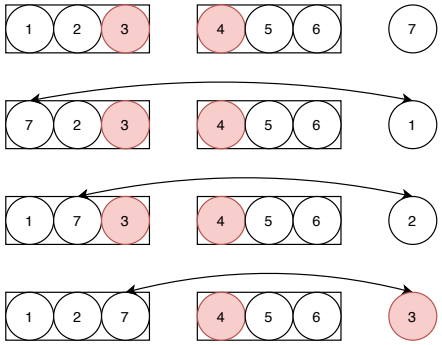

Figure 4.4: Bin rotation

Following $\frac{N}{I} < f+1$ we can not easily build $f+1$ distinct bins. However, there are $\frac{N}{I} > f$ processes, resulting in f distinct bins and $L = 1$ leftover processes.

Following this, as displayed in Figure 4.4, after constructing trees using the 2 bins of size 3, in the next step, we replace the members of one of the bins with the leftover process until a robust tree is found. Since there are $I = 3$ processes in the bin, in at most 3 additional steps ($\frac{I}{L}$ steps), we have found a robust tree.

Theorem 6 *An evolving graph $\mathcal{G}$ of generic trees satisfies linear Conformity if $\frac{N}{I} > f$.*

Proof 6 *We prove this by contradiction. Assume it is not possible to construct a robust graph if $\frac{N}{I} > f$. Given $\frac{N}{I} > f$, we know we can divide all processes into f distinct bins where at least one process is left over. Thus, this approach may only fail if there is either more than one faulty process in any of the bins or if there are faulty processes among the leftover nodes. However, since we divide all processes into f distinct bins, either one of these bins has to result in a robust graph (in f or less steps), or there is at most 1 faulty process in each bin (worst case). Thus, trivially, if none of the bins results in a robust tree, none of the leftover processes may be faulty. Following that, since $\frac{I}{L}$ is significantly smaller than N, trivially, this scheme fulfills Linear Conformity.*

The above approach can be even further relaxed to $\frac{N}{I} + L \geq f+1$ as long as $\frac{N}{I} \geq L$. This process is quite similar to the previous approach. However, it allows leveraging multiple leftover replicas L in a neat fashion.

The condition implies that for once, there are not more leftover processes than bins and, that the sum of leftover processes and bins is larger than the number of failures. Simplified, this means that, even in the worst case, there is at least one bin with at most one faulty process, at least one leftover process that is not faulty, and sufficient leftover processes L to apply each process in L to at least enough bins to reach the bin with at most one faulty process.

An example of this is displayed in Figure 4.6 for the tree in Figure 4.5. There is a total of $N = 10$ processes where according to $N = 3f+1$, there are at most $f = 3$ faulty processes and a total of $I = 4$ internal nodes. Following $\frac{N}{I}$ there are $\frac{N}{I} = 2$ bins of size $I = 4$ and $L = 2$ leftover processes. Since there is at least one faulty process per bin, after

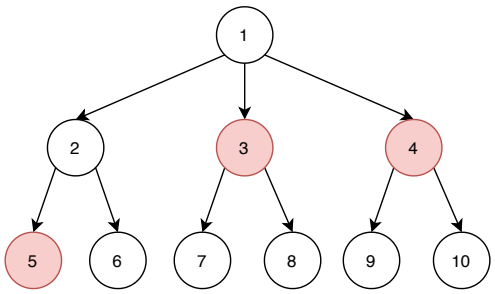

Figure 4.5: Generic Tree

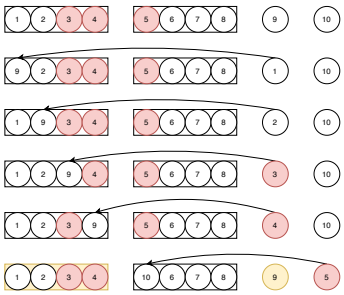

Figure 4.6: Bin rotation

reconfiguring the tree twice (once for each bin), it is impossible to build a robust tree. Thus, we combine the leftover processes in L with each bin. First, we replace each node in the first bin with the first leftover process. If this was unsuccessful, we know that either there must be two faulty processes in this bin or that the leftover node is faulty. However, this implies that according to the maximum number of faults, there is at most one faulty process in the remaining group, and the remaining leftover process must also be correct. Therefore, in the final step, after rotating the remaining leftover process over the remaining bin, a robust tree is guaranteed.

This algorithm always runs in linear steps as each leftover process in L is at most rotated over one bin each. As such, the complexity is the same as rotating one process over all processes in N. Resulting in $\frac{N}{I} + N$ steps which is still $\mathcal{O}(N)$.

Theorem 7 *An evolving graph $\mathcal{G}$ of generic trees satisfies Linear Conformity if $\frac{N}{I} + L \geq f + 1$ as long as $\frac{N}{I} \geq L$.*

Proof 7 *We prove this by contradiction. Assume it is not possible to construct a robust graph under this condition. Based on the algorithm, if there is a bin with at most one faulty node and there is one correct leftover node, we can combine the two to construct a robust tree. As such, essentially, we require two faulty processes for each and every pair of bin and leftover nodes. Either each bin has two faulty nodes, or the bin has one faulty node, and the leftover process is faulty.*

However, based on the restriction $\frac{N}{I} + L \geq f + 1$, there is indeed at least one bin with at most one faulty process and at least one correct leftover node. Thus, the only leftover possibility is that there are more leftover nodes than bins, and we ran out of replacement opportunities. However, this leads to a contradiction due to our second restriction $\frac{N}{I} \geq L$. As such, we have proven that this algorithm will achieve robustness.

Proving the linearity of the algorithm is trivial. We replace each node in each bin with exactly one leftover node. As such, the number of attempts never exceeds $\frac{N}{I} + N - L$ combinations.

The above leads to an interesting insight. $\frac{N}{I} + L \geq f + 1$ with the condition of $\frac{N}{I} \geq L$ can be simplified to $\frac{N}{I} + \frac{N}{I} \geq f + 1$. If we insert $N = 3f + 1$ again this results in $\frac{3f+1}{I} + \frac{3f+1}{I} \geq f + 1$ which is equal to $\frac{6f+2}{I} \geq f + 1$ where $I = \frac{6f+2}{f+1}$ which for $\lim_{f \to \infty}$ stays strictly below 6.

Therefore, for $N = 3f + 1$ by using this approach it is not possible to build a robust tree in linear steps with more than 5 internal nodes (including the root).

Further, we observe that in all of our linear approaches, f stays strictly below $\frac{N}{I} * 2$. We prove next that this is indeed the minimum condition for any linear reconfiguration strategy.

Theorem 8 *It is not possible to design Linear Reconfiguration strategies for $f > \frac{N}{I} * 2$ given the Strong Robustness criteria.*

Proof 8 *We prove this by contradiction. $f > \frac{N}{I} * 2$ implies that by dividing all processes into $\frac{N}{I}$ bins, we cannot tolerate more than two faulty processes per bin and still reconfigure the topology to a robust tree in linear steps. Let us assume that it is possible to tolerate two faulty processes per bin. Without leftover processes, the only possible strategy is to combine the processes of two distinct bins. However, even if there were precisely two faulty processes per bin, this requires a quadratic number of combinations and would lead to a contradiction.*

Thus, we do require leftover processes. As shown earlier, with one leftover process per bin ($L = \frac{N}{I}$), either the bin has two faulty processes or the respective leftover process is faulty. Following this, if we have one additional leftover process, we may iterate over all bins (N steps) to detect that each bin indeed must have two faulty processes, and, as a result, all the leftover processes must be correct.

Thus, we may attempt to build an additional bin using the leftover processes and processes from a distinct bin (given the knowledge that there is at least one correct process in each bin). However, this only works under two conditions. First, there may be at most one process missing to build an additional bin (else this would take quadratic steps), and second, none of the leftover processes may be faulty (else this results in a quadratic number of combinations again). While we may fulfill the first condition, the second leads to a contradiction. Following $f > \frac{N}{I} * 2$, there is actually one additional faulty process which means that we cannot guarantee any of the leftover processes to be correct and thus, require at least a quadratic number of steps.

4.3.3.4 Polynomial Conformity

As proven above, there are no linear reconfiguration strategies for $f > \frac{N}{I} * 2$. Therefore, we require approaches with higher complexity to find robust configurations for larger numbers of internal nodes.

Analogous to previous solutions, we divide all processes into a set of $\frac{N}{I}$ bins and then attempt to find a set of correct processes (one or more) to combine this set with each bin until a robust configuration is found. Trivially, if at least one bin has at most one failure, by combining one correct process with each of the bins, this eventually results in a robust configuration after at most N steps. More generically, if at least one bin has at most x faulty processes, we have to find x correct processes to combine them with each bin to find a robust configuration.

Since we know from the previous section that we may find a group of up to $x = 5$ correct processes in $\mathcal{O}(N)$ steps, we re-use this approach and combine the resulting group with all $\frac{N}{I}$ bins up to I^x times($N * \frac{N}{I} * I^x$).

This results in a complexity of $I^{x-1} * N^2$ which, since I is constant, can be simplified to $\mathcal{O}(N^2)$, thus, resulting in an upper limit of $f = (\frac{N}{I} * x) - 1$ faulty processes. Considering $N = 3f + 1$, that is $I = \frac{3xf+1}{f+1}$. Following this, considering up to $x = 5$ processes, we may find a robust configuration for up to $I = 14$ internal nodes in $\mathcal{O}(N^2)$ steps.

However, as mentioned, this exact approach only works for $x \leq 5$. Nonetheless, we now know that we may build a robust tree for up to $I = 14$ internal processes in N^2 steps. Thus, we may also find a robust configuration for $x = 14$ in $N^2 * I^x * \frac{N}{I}$ steps ($\mathcal{O}(N^3)$).

Generalizing this, assuming a total of $f = \frac{N-1}{3}$ faulty processes, there is at least one bin with at most $\lfloor \frac{I}{3} \rfloor$ faulty processes. Based on this, for any I we must find a solution for $\lfloor \frac{I}{3} \rfloor$ and use it to combine the $\lfloor \frac{I}{3} \rfloor$ correct processes with each of the $\frac{N}{I}$ bins. This way, for $\mathcal{O}(N^x)$ we may have up to $I = 1 + \sum_{n=0}^{x}(3^n)$ internal nodes. For this $x = log_3(-1 + 2i) - 1$ which results in the total number of steps of $N^{\lceil log_3(-1+2I)-1 \rceil} * I^{\lfloor \frac{I}{3} \rfloor} * \frac{N}{I}$ which has a complexity of $\mathcal{O}(N^{log(I)})$.

However, this only works if there are at least enough total processes N to build two distinct bins ($\frac{N}{I} \geq 2$).

Theorem 9 *For any I it is possible to find a robust configuration in at least $\mathcal{O}(N^{log(I)})$ steps as long as $\frac{N}{I} \geq 2$.*

Proof 9 *We prove this by contradiction. Assuming there is an I for which the above algorithm can not be used. This is only possible if there is no solution for $\lfloor \frac{I}{3} \rfloor$ which in itself is only possible if there is no solution for $\lfloor \frac{I}{9} \rfloor$. However, in Theorem 7 we already have proven that there are linear solutions for up to $I = 5$, and by dividing I consecutively by three, we eventually will reach $I \leq 5$ leading to a contradiction. Thus, the only remaining possibility is that there is a I for which the complexity is above $\mathcal{O}(N^{log(I)})$. However, since there is a linear solution for $I = 5$ and each fraction of $\frac{I}{3}$ of I results in an additional combination of the previous result with each bin ($\frac{N}{I}$ bins), the complexity indeed grows in a $log_3(I)$ fashion, thus resulting in $\mathcal{O}(N^{log(I)})$. Thus, also leading to a contradiction. Therefore, the only remaining possibility is that there is only one bin, and there are not sufficient processes to combine with this bin to exclude the faulty processes. However, by requirement, $\frac{N}{I} \geq 2$ leads to a final contradiction, proving this theorem.*

4.3.3.5 Practical Strongly Robust Trees

In this section, we outline the different upper limits given specific approaches considering the Strong Robustness criteria considering $f = \frac{N-1}{3}$ failures. Based on the described approaches, we can calculate the trade-off between the maximum number of internal nodes compared to their inherent reconfiguration complexity.

I	*Steps*
2	$f+1$
3	$\frac{N}{I}+f+1$
5	N
14	N^2
41	N^3
122	N^4
$\frac{N}{I} \geq 2$	$N^{\lceil log3(-1+2I)-1 \rceil}$
$\frac{N}{I} < 2$	$N!$

Table 4.1: Trade-offs of the different approaches

Due to this, in order to achieve Optimal Conformity while respecting $f = \frac{N-1}{3}$, there might be at most two internal nodes, including the root. An example tree that would roughly distribute the load equally among the two internal nodes is shown in Figure 4.7.

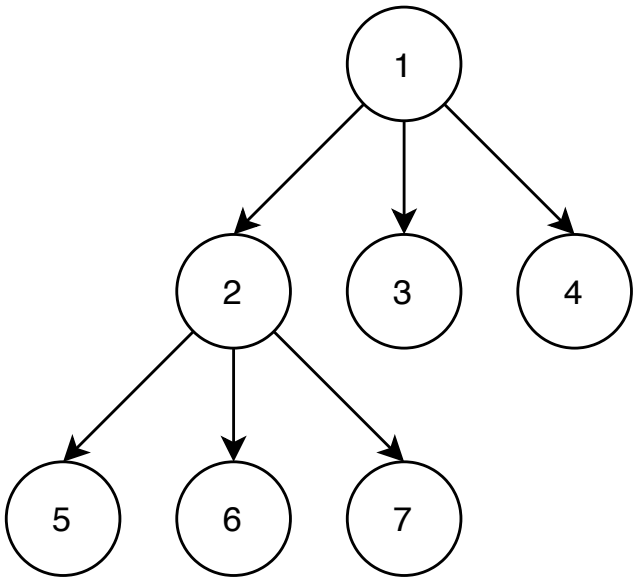

Figure 4.7: Minimum Tree Example

Slightly relaxing the conformity, subjecting the system to another $\frac{N}{I}$ reconfiguration steps allows to increase the number of internal nodes to 3. Further sacrificing Optimal Conformity while maintaining linear reconfiguration steps allows increasing the maximum number of internal nodes to 5, the upper limit for linear approaches (as proven in the previous section). As shown in the table, finding a robust tree within a viable number of steps becomes increasingly computationally complex for larger I. This leaves us with an interesting trade-off. For certain tree sizes we can either achieve Optimal Conformity or balance the load over a large number of internal nodes.

Considering the Strong Robustness criteria and $f = \frac{N-1}{3}$ failures, a tree might at most have two internal nodes. While this allows us to cut the cost of the star topology in half, it is not enough to achieve high scalability.

The following section discusses other possibilities for constructing robust trees in

optimal steps while also fully leveraging the tree structures' inherent load balancing properties.

4.3.4 Robust Trees

In the algorithm and description above, we assumed a worst-case scenario where given a tree of N nodes and I internal nodes at most $f_a < \frac{N}{I}$ nodes may be faulty to achieve optimal reconfiguration. This resulted from our Strong Robustness assumption that requires a path of correct processes between any correct processes in the system. This requirement comes from the worst-case assumption where, in the presence of $f = \frac{N-1}{3}$ faults, one faulty internal node that is a parent of at least one correct internal node already prevents consensus. As such, we concluded that no internal node must be faulty. Thus, as proven in the previous section, for I internal nodes, $\frac{N}{I} - 1$ presents the upper limits of failures if we want to reach a configuration without faulty internal nodes in optimal steps.

However, as soon as $f_a < \frac{N-1}{3}$, this is not necessarily true anymore. Considering $f_a = \frac{N-1}{3} - 1$ faults, one faulty internal node may hide one correct node and a quorum may still be collected. This observation leads us to a series of questions:

- How many correct processes have to be connected through a path of correct processes to collect a byzantine quorum;
- How many correct processes may a faulty internal node prevent from participating in the consensus;
- How many faulty internal nodes may exist for different sizes of f without preventing the collection of a correct quorum.

The first question is trivial to answer, considering the upper limit of failures $f = \frac{N-1}{3}$, we need a quorum of $|Q_{min}| = N - f$ correct processes.

The second question is more complex. Similar to the previous algorithm, assume a balanced m-ary tree. Given a tree of arbitrary depth, we can calculate the number of nodes one faulty internal node (excluding the root) in the worst case may hide by taking the upper limit of the division of the number of leaf nodes and the fanout resulting in $\lceil \frac{N-m-1}{m} \rceil$.

Finally, before answering question three, let us re-capture the previous reconfiguration algorithm. All nodes N are divided into $\frac{N}{I}$ disjoint bins. Considering $f_a < \frac{N}{I}$ faults, at least one of these bins has no faulty nodes. Therefore, if we attempt to build a tree where each of the bins represents the set of internal nodes, we will eventually pick the bin with no faulty nodes and build a tree without faulty internal nodes and, as previously established, a tree without faulty internal nodes is trivially robust.

In the next step, we relax the assumption of the algorithm. Instead of $f_a < \frac{N}{I}$ we assume, $f_a < 2\frac{N}{I}$. In this setting, if we divide the faulty nodes equally among the bins as in the previous algorithm, while each bin in the worst case has one or more faulty nodes,

there is guaranteed at least one bin with at most one faulty node. Under the assumption that a tree with one faulty internal node might be robust, this is only possible if the faulty node is not at the root of the tree. Thus, we have to adjust the algorithm slightly. Instead of constructing one tree per bin, we construct two trees per bin but alter the selected root process. Therefore, by constructing two trees per bin, considering that at least one bin has at most one faulty process and we can tolerate one faulty internal node as long as it is not the root. After $2\frac{N}{I}$ attempts, we will construct a robust tree. As we have doubled both the number of reconfiguration steps and the number of tolerated faults, this algorithm still terminates in optimal steps.

However, we have not yet calculated exactly how many internal nodes might be faulty. To answer this question, we start with an example. Assuming a system of $N = 421$ Nodes with a fanout of $m = 20$, depth $d = 2$ and a minimum quorum size $|Q_{min}| = 281$. Assuming $f_a = 1$ faulty internal node, at most $m = 20$ correct nodes might be hidden by this 1 faulty internal node. Given the adjusted algorithm, if we tolerate 1 faulty internal node $f_a < 2m$ and as such, in the worst case, $f = 39$. Thus, by adding the 39 arbitrarily distributed faults in the tree structure to the 20 correct nodes that might be blocked by the 1 faulty internal node, 59 nodes are unable/unwilling to contribute to the quorum. If we subtract this from the $N = 421$ nodes that results in $|Q| = 362$ nodes which is larger than the minimum quorum size $|Q_{min}| = 281$. As such, in this example, our system can tolerate one faulty internal node, and as a result, duplicate the number of tolerated faults compared to our previous algorithm.

This process does not change for deeper trees, as, in the worst case, we have to assume that the faulty nodes are in the highest positions in the tree and, as such, one faulty internal node may prevent at most $\lceil \frac{N-m-1}{m} \rceil$ correct processes from participating.

We generalize the above process by assuming a balanced tree of N nodes with a fanout of m. From above, we know that the number of nodes that are prevented from participating in consensus is bound by the number of faulty internal nodes. Given f_a failures and $\frac{N}{I}$ bins, we know there is a bin with at most $\lfloor \frac{f_a}{\frac{N}{I}} \rfloor$ faulty internal nodes.

Consider $\lceil \frac{N-m-1}{m} \rceil$ to be the number of nodes a faulty internal node can prevent from participating (i.e., all its children and children to children down to the leaves). As such, at most $\frac{f_a}{\frac{N}{I}} * \lceil \frac{N-m-1}{m} \rceil$ nodes can be excluded from consensus this way.

In addition to that, in the worst case, f_a faulty nodes are children of correct nodes (either faulty internal nodes or leaf nodes of correct internal nodes) which also will not participate in consensus. Thus, as long as the sum of these two factors $\frac{f_a}{\frac{N}{I}} * \lceil \frac{N-m-1}{m} \rceil + f_a$ is lower than the maximum number of failures f, a quorum can be obtained. Therefore, as long as $\frac{f_a}{\frac{N}{I}} * \lceil \frac{N-m-1}{m} \rceil + f_a \leq \frac{N-1}{3}$ consensus can be achieved.

Theorem 10 *We can achieve consensus as long as* $\frac{f_a}{\frac{N}{I}} * \lceil \frac{N-m-1}{m} \rceil + f_a \leq \frac{N-1}{3}$ *and the root is a correct process*

Proof 10 *We prove this by contradiction. Assume that no byzantine quorum could be collected.*

Algorithm 6 Construction of a Robust Tree

```
1: function INIT(𝒩, m) ▷ Initialize the evolving tree with the set of nodes 𝒩 and fanout m
2:     B ← ∅                                              ▷ Initialize the set of bins
3:     for all i ∈ 𝒩 do                                   ▷ For each Node
4:         B^{i mod m} ← B^{i mod m} ∪ i                  ▷ Assign the node i to one of the m bins
5:     end for
6: end function
7: function BUILD(k)
8:     i ← k mod m
9:     𝒢^i ← all possible trees whose internal nodes are drawn exclusively from B^i.
10:    r ← B^i[⌊k/m⌋]                                     ▷ Pick Root Process from B^i
11:    T_r^k ← pick any tree at random from 𝒢^i with root r
12:    return T^k
13: end function
```

Because of $\frac{f_a}{\frac{N}{I}} * \lceil \frac{N-m-1}{m} \rceil + f_a \leq \frac{N-1}{3}$, *we know that either* $f_a < \frac{N}{I}$ *and according to Theorem 5 we can achieve consensus in optimal steps or* $f_a \geq \frac{N}{I}$ *and hence, in the worst case, there might be one or more faulty nodes in each bin. As we draw the internal nodes of a given tree exclusively from a single bin, this means that we might have faulty internal nodes. Due to this, consensus may be prevented as faulty internal nodes might block correct nodes (their children) from participating in consensus.*

Consider the case where the root is correct and one or more internal nodes are faulty. Note that each faulty internal node may prevent at most $\lceil \frac{N-m-1}{m} \rceil$ *(all its children) from participating in consensus. As nodes are equally distributed over all* $\frac{N}{I}$ *bins, there is at least one bin with at most* $\lfloor \frac{f_a}{\frac{N}{I}} \rfloor$ *faulty processes. Thus, in a tree constructed from this bin, at most* $\lceil \frac{N-m-1}{m} \rceil * \lfloor \frac{f_a}{\frac{N}{I}} \rfloor$ *processes may be prevented from participating in consensus, in addition to the* f_a *faulty nodes themselves. As such, as long as* $\lceil \frac{N-m-1}{m} \rceil * \lfloor \frac{f_a}{\frac{N}{I}} \rfloor + f_a \leq \frac{N-1}{3}$ *we can collect a quorum and solve consensus. Thus, as long as the root is a correct process a quorum may be collected process is correct, which leads to a contradiction.*

While Theorem 10 proves the upper limit of failures our tree structure can tolerate, it still assumes a correct node at the root. Thus, to achieve this, we adjust Algorithm to account for that.

Algorithm 6 shows the adjusted algorithm in this regard. As such, in addition to picking each bin round-robin, for each bin, we construct trees with different root nodes until reaching a robust tree.

Theorem 11 *We can achieve optimal reconfiguration for* $\frac{f_a}{\frac{N}{I}} * \lceil \frac{N-m-1}{m} \rceil + f_a \leq \frac{N-1}{3}$.

Proof 11 *We prove this by contradiction. As Algorithm 6 constructs several trees per bin, where each tree has a different root node, it eventually reaches a correct root node. And, as proven in Theorem 10 as long as* $\frac{f_a}{\frac{N}{I}} * \lceil \frac{N-m-1}{m} \rceil + f_a \leq \frac{N-1}{3}$ *and the root node is correct, consensus can be achieved. Assume that Algorithm 6 requires at least* $f_a + 2$ *reconfiguration steps.*

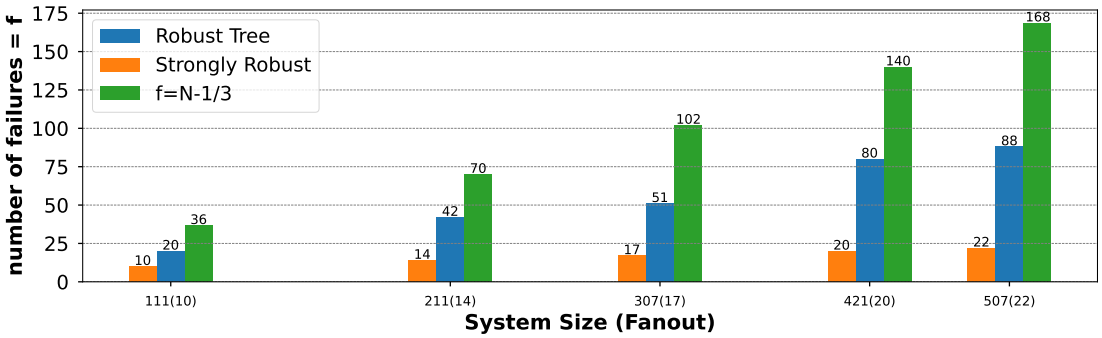

Figure 4.8: Comparison of different scenarios.

As nodes are equally distributed over all $\frac{N}{I}$ *bins, there is at least one bin with at most* $\lfloor \frac{f_a}{\frac{N}{I}} \rfloor$ *faulty processes. In the worst case, this is the last bin, i.e. bin* $\frac{N}{I}$*. Algorithm 6 iterates over all* $\frac{N}{I}$ *bins in sequence and rotates the root until a correct configuration is reached which requires at most* $\lfloor \frac{f_a}{\frac{N}{I}} \rfloor * \frac{N}{I} + 1$ *steps. This leads to a contradiction as* $\lfloor \frac{f_a}{\frac{N}{I}} \rfloor * \frac{N}{I} + 1$ *is never bigger than* $f_a + 1$*.*

4.3.4.1 Realistic Scenarios

Based on the previous equation we can calculate the upper limit of failures f_a a given tree structure can tolerate based on the number of nodes N and the fanout m.

Figure 4.8 shows the number of failures we may tolerate based on the equation and compares it to the simplified algorithm and the optimal number of failures f. We vary the fanout and system size on the x-axis and show the number of failures the algorithm may tolerate on the y-axis. We see that the robustness of the extended algorithm grows with increasing system size contrary to the simplified algorithm. In this regard, our proposed algorithm allows to maintain our tree structures for roughly more than half of the worst case number of failures.

4.3.5 Gracefully Degraded Reconfiguration

Algorithm 6 only guarantees the construction of a robust tree as long as $\frac{f_a}{\frac{N}{I}} * \lceil \frac{N-m-1}{m} \rceil + f_a \leq \frac{N-1}{3}$. We describe this upper bound of failures as f_a^{max}. Thus, Algorithm 6 is not able to guarantee the construction of a quorum robust tree for any $f_a > f_a^{max}$.

However, this is only true in the worst case when faulty nodes are distributed equally over all bins. In practise, unless the adversary can control the distribution of faulty nodes over the bins, this is unlikely. As such, it might still be possible to construct a robust tree for $f_a > f_a^{max}$ as long as there is a bin with at most $\lfloor \frac{f_a^{max}}{\frac{N}{I}} \rfloor$ nodes. Under this assumption, we're always guaranteed to find a robust tree in $f_a^{max} + 1$ steps (see Theorem 10).

We model this probability as an Urn problem [4]. In detail, there are N nodes, out of which f are faulty and $N - f$ are correct. For $\frac{N}{I}$ rounds (the number of bins) we draw I nodes (the number of nodes in each bin) without replacement and calculate the

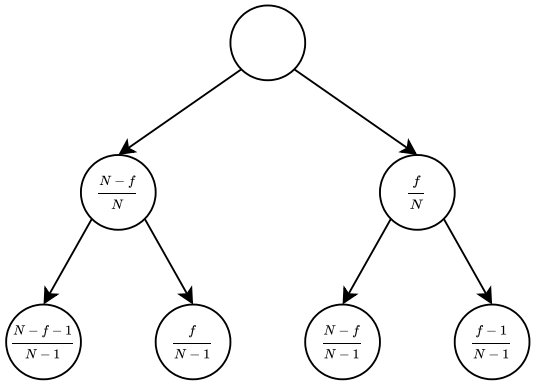

Figure 4.9: Example Probability Tree

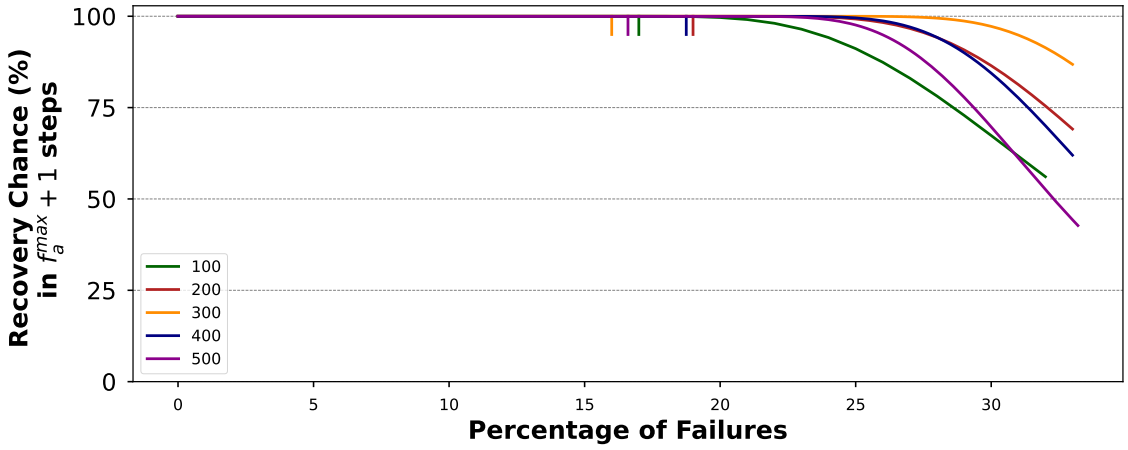

Figure 4.10: Probability of finding a quorum robust tree for an increasing number of faults. The vertical bars for each system size delimit f_a^{max} up to which Algorithm 6 ensures the construction of such a tree.

probability of picking at most $\lfloor \frac{f_a^{max}}{\frac{N}{I}} \rfloor$ faulty nodes (the number of faulty internal nodes we can tolerate and still achieve consensus following Theorem 10). In this context, each reconfiguration attempt follows a hypergeometric distribution [55].

If we sum up the probability of all successful branches in a probability tree based on the above scenario, this translates to the estimated probability of finding a robust tree for any f_a. Each pick from the set of nodes represents 2 points in the probability tree, one representing the probability of picking a correct node and one the probability of picking a faulty node. An example of such a probability tree is presented in Figure 4.9.

Figure 4.10 depicts the probability of finding a quorum robust tree for different system sizes N (100, 200, 300, 400, 500) and their respective fanout m (10, 14, 17, 20, 22) with an increasing number of faults. Where we got the probability to find a robust tree on the y-axis for a given percentage of failures f on the x-axis. Interestingly, our proposed algorithm has a very high chance to find a robust tree, independently of the system size up to $\frac{1}{4}N$. However, even at $f_a = \frac{N-1}{3}$ failures, most configurations have a higher than

50% chance to construct a robust tree in $f_a^{max} + 1$ steps.

As such, we conclude that not only does the extended approach guarantee better resilience in the worst case, as shown in Figure 4.8 but also, in the average case, has a substantial chance to find a robust tree at any system size and number of failures. In the case we were not able to achieve consensus after f_a^{max} reconfigurations we can still fall back to a minimum tree (as discussed in Section 4.3.3.5) and still preserve some of the advantages of using a tree structure. As such, in the worst case we might have to reconfigure $f_a^{max} + f + 1$ times until reaching a robust structure.

4.4 Pipelining

As mentioned before, due to the additional communication steps in the tree, larger latencies significantly impact the overall system throughput. i.e., in a tree of depth three, instead of one communication step per consensus phase, compared to PBFT, there are six communication steps. Applied to a geo-distributed system with a roundtrip latency of 200ms, most processes will remain idle for over a second between each block proposal. As such, this essentially results in an upper limit of how much throughput a system can provide.

HotStuff deploys a simple pipelining scheme, where multiple consensus phases are piggybacked on one another. As a result, it compensates for the additional consensus phase compared to PBFT and achieves high throughput in many settings. A more detailed explanation of HotStuff's Pipelining can be found in Section 3.

While piggybacking allows to reduce the performance impact of multiple phases, it does not compensate for additional communication steps. As such, it is possible to apply HotStuff's Pipelining scheme to PBFT, which would give PBFT a considerable performance advantage in a high latency setting.

4.4.1 Pipelining Stretch

Due to this reason, we introduce the *Piplining Stretch*. We call it a pipelining stretch as it represents an additional pipelining layer on top of the inherent pipelining HotStuff already offers.

While, in theory, this may also be applied to HotStuff, in practice, as we will show in the following sections, the potential speedup is very limited due to the inherent scalability issues of HotStuff.

In a nutshell, in HotStuff, in each given round, the leader has to disseminate N Messages and then receive and process N Votes. Thus, the leader is busy for a long time, which is further aggravated at larger system sizes (larger N). Meanwhile, in a tree, each process (including the leader) has to disseminate at most m messages and process at most m votes. Thus, not only will each process be significantly busy for a shorter time period,

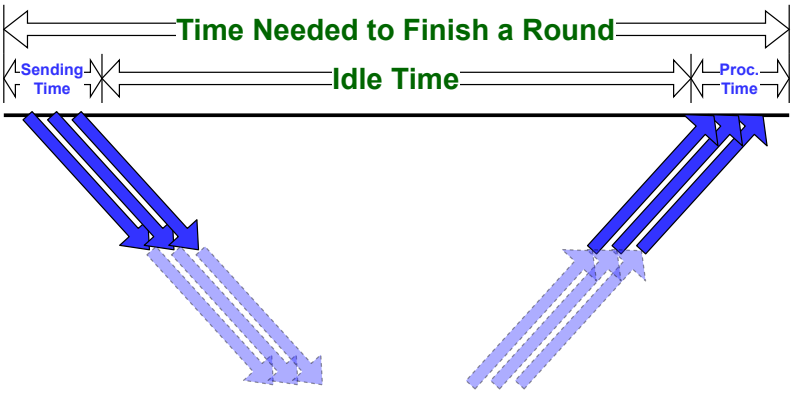

Figure 4.11: Idle Time in Kauri

but, due to the additional communication steps, between a given "busy period", there are significantly longer idle times.

We show this in Figure 4.11. Given the total time it takes to finish a round, there are two periods during which a process is busy. For once, there is the *sending time*.

Sending time The sending time describes the time a node is busy disseminating the block to its children in the tree (i.e., the root at the beginning of a round and the remaining processes when relaying the block). Three main factors influence the sending time. First, and most importantly, the bandwidth, trivially, the smaller the bandwidth, the longer it takes to disseminate data. Then, the fanout and block size, the larger the fanout or block size, the more data has to be disseminated, occupying more bandwidth and preventing the system from moving to the next round. In the course of this thesis, we present this as $\frac{mB}{b}$ where m is the fanout, B is the block size, and b is the link bandwidth.

Besides the sending time, there is also the *processing time*.

Processing time The processing time is the time during which a process verifies and aggregates the signatures it received from its child processes. The processing time depends on the fanout m and the average processing time per signature Φ. As such, we present this as $m * \Phi$.

As such, if we take the total time and subtract both sending and processing time, we get the *idle time*.

Idle time The idle time represents the time after a block was successfully disseminated and before receiving signatures to verify and aggregate. It strongly depends on the round trip time, depth of the tree, and processing delay at each of the intermediary hops. We describe it through:

$$\textit{Idle time} = h \cdot (RTT + \textit{processing time})$$

for a tree of height h and the round-trip latency RTT.

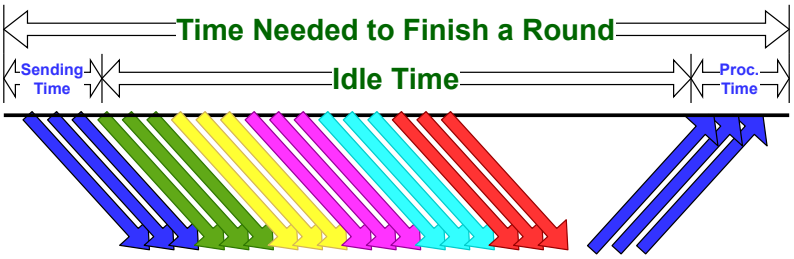

Figure 4.12: Pipelining in Kauri

The pipelining stretch leverages the above observation. Given there is an ample enough idle time, the leader can use the knowledge of the previous block b_i they proposed, and, tentatively, propose block $b_{i+1}, b_{i+2}, ..., b_{i+n}$ before receiving the consensus result for block b_i. This process is depicted in Figure 4.12 where each distinct arrow color represents a different block proposal. As we leverage the same pipelining as HotStuff, considering a pipelining stretch of n and a given tentative block b_i, the quorum signature of b_i is propagated with the next tentative block in round b_{i+n+1}. As such, instead of having each consecutive round carry the information of previous rounds, the information is always n blocks apart due to the stretch. As a result, in the presence of failures, similarly to HotStuff, all tentative blocks that have not received three consecutive quorums have to be aborted, and the pipeline must be rebuilt from scratch.

We can estimate the potential speedup based on the above premises. As, in practice, sending and processing may fully overlap without deterring each other, we assume that they are done concurrently.

Therefore, the number of concurrent instances that we may execute during the *idle time*, is approximately given by $\frac{idle\ time}{\text{MAX}(processing\ time, sending\ time)}$. Thus, if the system is bound by the bandwidth, the *sending time* will limit the potential speedup, and if the system is bound by the processing capabilities, it will be limited by the *processing time*.

Therefore, compared to a star, where the leader has to send and process $N-1$ messages, we can reduce the system load to m messages by using a tree. Based on that, as such, the maximum speedup in comparison to a star is bound by $\frac{N-1}{m}$ (e.g., for a system with $N = 401$, a tree with a fanout of $m = 20$, we can theoretically achieve a speedup of 20.

4.4.2 Practical Speedup Potential

While the maximum speedup of the system is $\frac{N-1}{m}$ compared to the star topology, the practical speedup varies in different scenarios. We measured the block size of the different approaches for a fixed batch size and the linear computation time required for each block.

Table 4.2 shows the experimentally measured Block Size and Computation time for

N	Block Size	Computation Time
BLS		
100	257 560b	17,13ms | 3,65ms
200	257 688b	22,88ms | 5,3ms
400	257 880b	27,37ms | 6,9ms
Secp256k1		
100	291 088b	3,96ms
200	325 520b	7,25ms
400	393 296b	15,02ms

Table 4.2: Measured values for Block Size and Computation time for different system sizes.

both the *Secp256k1* cryptographic algorithm and BLS. The base block size of the system is 256kb (i.e., 1000 operations per block). As such, everything above that represents the inherent overhead of each of the approaches. As BLS aggregates a set of signatures to a single signature, there is a negligible data overhead for larger system sizes as processes have to send identifying information of the signing processes alongside the block. In our implementation, this is done with the help of a bit array where each bit represents a single process. If the bit is "1", the process contributed their signature, else it is "0".

Secp256k1 signatures, on the other hand, do not support aggregation and, as such, they have a significant data overhead, almost doubling the block size at large system sizes as the full array of signatures has to be sent alongside the block to all processes.

We separated the computational load of BLS into two sub-categories. The first is "Verify and Aggregate", where we verify all incoming signatures, aggregate the signatures and then verify the aggregate, and the second is "Aggregate and Verify", where we aggregate all incoming signatures and then verify the aggregate subsequently. While the second is significantly faster (as shown in the table), if a process receives an invalid signature, the verification of the aggregate will fail, and the process has to verify each signature before constructing another aggregate resulting in a small overhead (i.e., the sum of both approaches) in the worst case. Nonetheless, this is an unlikely attack that can easily be mitigated, as due to the point-to-point channels, processes can quickly identify an adversary and exclude the faulty process from the aggregate in the next round.

While BLS signatures are significantly more expensive than Secp256k1, in a tree, we have to process at most m signatures on each level. As such, the measured value represents the cost for $m = 10, m = 14, m = 20$ for the different system sizes. Furthermore, as Secp256k1 does not support aggregation, its cost increases linear with the number of processes and does not allow the "Aggregate and Verify"optimization.

Table 4.3 summarizes this discussion. For this sake, we have created three different scenarios, namely "National", "Regional", and "Global", with increasing geographic distribution. "National"is a low latency (5ms) high bandwidth (1Gb/s) setting for a blockchain deployment within a single smaller country (Germany, France, Japan) or several interconnected data centers. "Regional"is a scenario spanning over a series of countries or states of

Scenario	N	Height h	Root's fanout m	Processing time	Sending time	Remaining time	Base pipelining	Pipelining stretch	Total depth	Expected speedup
HotStuff-secp										
National	100	1	99	4	29	14	4	1	4	-
National	200	1	199	7	65	17	4	1	4	-
National	400	1	399	15	156	25	4	1	4	-
Regional	100	1	99	4	288	104	4	1	4	-
Regional	200	1	199	7	648	107	4	1	4	-
Regional	400	1	399	15	1569	115	4	1	4	-
Global	100	1	99	4	1153	204	4	1	4	-
Global	200	1	199	7	2591	207	4	1	4	-
Global	400	1	399	15	6277	215	4	1	4	-
Kauri										
National	100	2	10	3,6	2,5	24	4	8	32	$\approx$ 8x
National	200	2	14	5,3	3,6	25	4	6	24	$\approx$ 12x
National	400	2	20	6,9	5,1	27	4	5	20	$\approx$ 22x
Regional	100	2	10	3,6	25,7	203	4	9	36	$\approx$ 11x
Regional	200	2	20	5,3	36,1	205	4	7	28	$\approx$ 18x
Regional	400	2	14	6,9	51,6	206	4	5	20	$\approx$ 30x
Global	100	2	10	3,6	103,0	403	4	5	20	$\approx$ 11x
Global	200	2	14	5,3	144,3	405	4	4	16	$\approx$ 18x
Global	400	2	20	6,9	206,3	406	4	3	12	$\approx$ 30x

Table 4.3: Pipelining stretch and estimated speedup vs HotStuff-secp for a block size of $250Kb$ and different N.

a larger country (US, EU, China) with 100ms round trip latency and 100Mb/s bandwidth. Finally, "Global"is a geographically distributed blockchain deployment used in several works in the literature [26, 36] with 200ms round trip latency and 25Mb/s bandwidth.

For each configuration, we provide the scenario, the number of processes, the depth of the hierarchy (i.e., 1 for stars, 2 for trees), and the fanout at the root ($N-1$ for stars and m for trees). Based on this, we filled in the measured values for the processing time, sending time, and remaining time. With the help of these variables, we can then calculate the pipelining stretch (the extension of the already existing pipelining of HotStuff), resulting in a total depth of $4 * stretch$.

The first thing to note is that independent of the scenario; it is impossible to deploy any additional pipelining in HotStuff due to the bottleneck at the root and the smaller latency. Meanwhile, by distributing the load in the tree, Kauri can leverage the remaining time in the system with a pipelining stretch up to 9.

This difference results in an expected speedup ranging, depending on the scenario, from 8 to over 30. Note that this speedup exceeds the theoretical maximum speedup of

$\frac{N-1}{m}$ we calculated earlier. This is the case, as not only is there a bottleneck at the leader, but, in addition to that, the Secp256k1 signatures used in HotStuff are not aggregated and require significantly more bandwidth than the BLS signatures.

We stress that the values in the Table 4.3 are not exhaustive and neither attempt to be, but rather represent a baseline of scenarios that were used in the context of this thesis (e.g., we also experimented with varying bandwidth, block-sizes, latency and deployed up to 800 processes). The general purpose of this table is twofold. For once, we want to clearly show that, while the pipelining stretch could theoretically also be applied to HotStuff and star topologies, in practice, this is not feasible as the inherent leader bottleneck and the lower latency leave insufficient remaining time for additional pipelining. On the other hand, it offers a sound theoretical basis for the results shown in the evaluation section.

4.5 Implementation

In this section, we discuss the implementation details of Kauri, our prototype, that was developed throughout this thesis. Kauri was implemented as an extension of the original HotStuff prototype [3].

The alterations of the original HotStuff implementations can be separated into four categories:

- BLS signature support
- Tree Construction Algorithm
- Tree dissemination and aggregation
- Pipelining

The source code of the prototype is available on Github[4]. The adaptations and additions contribute around ≈ 1400 lines of code to the original HotStuff codebase.

In the following sections, we describe in detail how each of the concepts was implemented.

4.5.1 BLS Signatures

The HotStuff prototype provides an abstraction that allows adding support for diverse cryptographic primitives. As such, we extended these abstractions to add support for the BLS cryptographic scheme. For this purpose we used the publicly available implementation used in the Chia Blockchain [11, 20][5]. We implemented the following abstractions:

- Public Key

[3] Available at https://github.com/hot-stuff/libhotstuff
[4] Available at at https://github.com/Raycoms/Kauri-Public
[5] Available at https://github.com/Chia-Network/bls-signatures

- Private Key
- Signature
- Certificate
- Quorum-Certificate

We used the Pop-BLS scheme provided by the library. While this is the most efficient aggregation scheme the library offers, it is vulnerable to rogue key attacks, making it unfeasible in permissionless blockchains. However, in our use case, in permissioned blockchains, we can prevent participants from arbitrarily changing their keys during operation.

Thus, with the help of BLS signatures its possible to aggregate signatures while they are being disseminated up the tree at $\mathcal{O}(m)$ cost at each internal node. Each signature aggregate can then be verified in $\mathcal{O}(1)$ steps. We can use the same scheme also with the star, requiring $\mathcal{O}(N)$ cost at the leader, but allowing verification in $\mathcal{O}(1)$ steps at all the other nodes. On top of that, it has a constant signature size independent of the number of processes N, which reduces the bandwidth cost significantly in comparison to the cryptographic scheme *secp256k1* that is used in HotStuff by default. The original "vanilla" implementation of HotStuff is therefore denoted as *HotStuff-secp* and the BLS variant as *HotStuff-bls*. We do this to show that the performance improvements are independent of the usage of the cryptographic scheme but are actually the unique combination of trees, signature aggregation, and optimistic pipelining.

In this context, we would also like to mention our contribution to the BLS codebase resulting in a significant speedup for public key aggregation [6].

4.5.2 Tree Construction & Reconfiguration

HotStuff already offers a mechanism to trigger reconfigurations after reaching a specific timeout. Then, the next leader is chosen deterministically and, after a short timeout, proposes a new block (extending the last locked block) to kickstart a new round of consensus. In Kauri, alongside the change of leader, each process calls the *build* primitive outlined in Algorithm 6.

The construction of the tree follows Algorithm 5. As we will not always necessarily have sufficient processes to construct a perfect m-ary tree, we equally distribute nodes among the internal nodes to achieve a balanced tree to maximize load distribution. As such, depending on the number of processes, the highest load will always be on the root. Given a fanout m, we construct a tree where each node has at most m child processes. Each process runs the same deterministic algorithm and obtains their child and parent processes as a result.

[6] Available at https://github.com/Chia-Network/bls-signatures/pull/136

In order to achieve disjoint sets of internal nodes after each failure, we rotate the entire set of nodes by the number of internal nodes. We do this for a total of $f_a + 1$ times. If after $f_a + 1$ attempts consensus was not achieved, we fall back to a star topology.

We define several timeouts in the system. The limit on message transmissions between two nodes δ after which a process moves on (as described in Section 4.1) and the absolute upper limit Δ. Based on the value of δ we calculate the *round termination timeout* as $\delta * depth * 2$ after which a reconfiguration is enacted. Every time a reconfiguration is enacted, δ is doubled until reaching or exceeding Δ at which it is capped.

4.5.3 Broadcast and Await

The core of this part is related to the implementation of both the *broadcast* and the *await* primitive, which is putting Algorithm 2 and Algorithm 3 in practice. Extending the HotStuff dissemination is trivial. Instead of broadcasting a block, the root process propagates the block only to their child processes (i.e., in a star $N-1$ processes, in a tree only m processes). Following that, on receiving a proposal, a process forwards it directly to their own child processes.

The implementation of await is a bit more complex. Internal nodes will now receive votes, process and aggregate them and then relay the vote to their parent. For this sake, we created a new message type: "vote-relay", and a handler for that.

4.5.4 Pipelining

The HotStuff paper describes the HotStuff pipelining as piggybacking additional information in each consensus message to process several phases in parallel. In practice, in the implementation, HotStuff produces a new block for each consensus round and considers a quorum on block i as an implicit second quorum for block $i-1$, etc. Thus, HotStuff finalizes a given block b_i after reception of a quorum for block b_{i+4}

We extend on top of this system and propose additional blocks optimistically following the approach discussed in detail in the previous section. In Kauri, the current pipelining assumes a static pipelining stretch throughout the experiment. To estimate this, we use the theoretical model outlined in Section 4.4 and the obtained measurements to calculate the pipelining stretch for a given scenario. The root process in the tree keeps track of all concurrent blocks at a given moment. The leader initially disseminates additional blocks until reaching the pipelining limit. After receiving a quorum for the next pipelined block, a new block is directly proposed to keep up the number of concurrent blocks at all times. Based on this, instead of finalizing a block after receiving a quorum for block b_{i+3}, for a given stretch s it takes b_{i+s*4}

In practice, the number of pipelined blocks should be adapted at runtime depending on the system's current state. i.e., when there is more bandwidth and computation power available, pipelining should be increased, while during periods of lower load or increasing

latency, pipelining could be reduced to preserve resources and adapt to the load. However, this is more complex than initially meets the eye and is left open for future work.

4.6 Summary

This chapter introduced our proposed strategies to overcome the bottlenecks and hindrances of building a highly performant and scalable byzantine fault tolerant consensus protocol. We first outlined how tree structures compare to star topologies and how we can leverage the similarities to offer the same guarantees. Next, we discussed the shortcomings of tree structures and our novel approaches to pipelining and reconfiguration to solve them. Then, we introduced a performance model to reason about the performance we can achieve with the help of our strategies. Finally, we outlined the implementation details related to our developed prototype that combines our proposed strategies. The experimental evaluation of this prototype is presented in the next chapter.

5

Evaluation

In this chapter, we discuss the experimental evaluation of the developed prototype. We analyze the throughput and latency of our approach in comparison to state-of-the-art approaches like HotStuff.

In detail, we start by evaluating the performance improvement from the Aggregate-Verify scheme. We then evaluate how the performance develops for different pipelining stretches considering different block sizes. Next, we show how Kauri behaves under changing round trip times. This brings us to the central part of the evaluation, evaluating the throughput of Kauri in different scenarios with 100 to 800 processes and how Kauri scales independent of the number of processes by leveraging deeper trees. Following that, we analyze how the latency of Kauri develops for different block sizes and bandwidths. Finally, we evaluate how Kauri behaves in a heterogeneous deployment, and last but not least, we show the behavior of Kauri in the presence of different failure scenarios.

5.1 Experimental Setup

We executed all experiments on the Grid'5000 testbed [8] using up to 80 physical machines where each physical machine has two Intel Xeon E5-2630 v3 8-core CPUs and 128 GB RAM (Paravance and Parasilo Clusters in Rennes) where the physical machines are connected through a 10 Gb Network. In each of the experiments, we deploy at most 10 virtual processes (Docker containers) per physical machine, thus, reaching up to 80 physical machines for the experiments with 800 processes.

As already pointed out before, we created three main scenarios that we use to evaluate Kauri. These scenarios are based on actual use-cases and practical application scenarios. Those scenarios are, *global*, *regional*, and *national* described in Section 4.4.2.

In addition to these homogeneous scenarios, we also considered one heterogeneous deployment scenario with variable bandwidth and round trip time, which we obtained from recent work [58] that measured latency and bandwidth for a set of globally distributed clusters.

In order to implement the network characteristics, we have deployed NetEm [32]

N	100	200	300	400	500	600	700	800
m	10	14	17	20	22	24	26	28

Table 5.1: Fanout m for a selection of system sizes N.

network latency and bandwidth restrictions on each individual docker container in the case of the homogeneous deployments and, for the heterogeneous setup, we have used the network emulation tool Kollaps [29].

In all experiments, we deployed a round number of processes ($N = 100, 200, 300, ..$), which typically does not result in perfect m-ary trees. This strategy also matches a real-world deployment more closely. Thus, to construct the tree, we simply start with a fanout of m at the root and then equally distribute the remaining nodes over the internal nodes in the tree to construct an approximately balanced tree. In most experiments, unless otherwise stated, we have used a tree of depth two and constructed the tree with the smallest possible root fanout for the given scenario. The resulting fanout m for given system sizes N is outlined in Table 5.1.

Besides the two variants HotStuff-bls and HotStuff-secp, we have also executed a subset of the experiments with Kauri without pipelining, which approximates the implementation of Motor [36]. Hereafter, we refer to this deployment as Motor*. As we pointed out in Section 3, Motor uses a tree of depth two without pipelining and a similar cryptographic scheme as used in Kauri. In the absence of failures, Kauri without pipelining performs very similar to Motor. We used this, as the original prototype of Motor is not publicly available. Furthermore, while Motor only has 2 phases of consensus, our implementation of Motor uses the HotStuff pipelining to balance that out.

5.2 Preliminary Experiments

We first take a look at a row of preliminary experiments, particularly at how the improved Aggregate & Verify scheme performs and how pipelining in practice compares to our theoretical model.

5.2.1 Aggregate & Verify

First, we take a look at the quantitative improvements resulting from the improved signature verification strategy. As discussed in Section 4.4.2, it has no negative drawbacks as the only potential attack vector may be mitigated as correct processes may detect incorrect processes easily and ignore their input in the following round.

We have executed an experiment in all three of our scenarios (global, regional, and national), considering 100 processes with both strategies and comparing their throughput and CPU load on the respective machines.

The results of this experiment are displayed in Figure 5.1 comparing the throughput for different approaches and in Figures 5.2, 5.3, 5.4 comparing the respective load at each

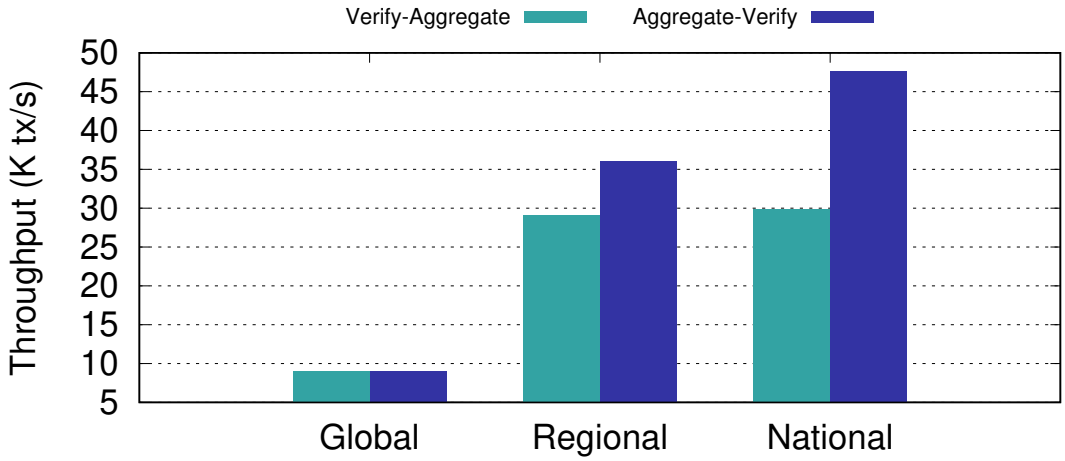

Figure 5.1: Throughput of the two different signature verification strategies.

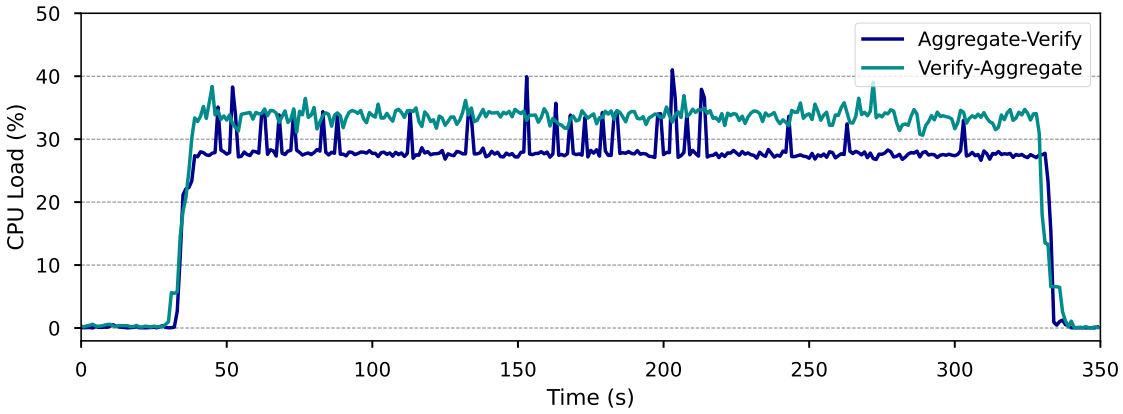

Figure 5.2: CPU Usage: national (10*ms* RTT - 1*Gb/s* links)

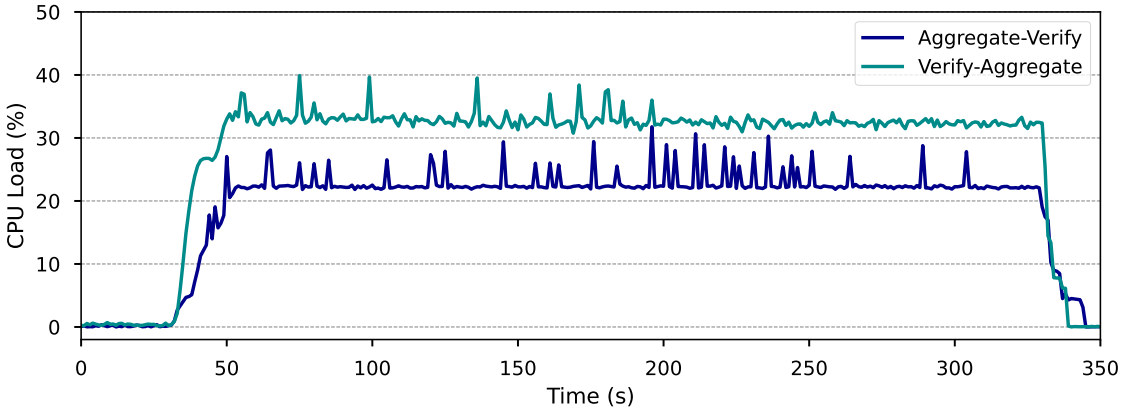

Figure 5.3: CPU Usage: regional (100*ms* RTT - 100*Mb/s* links)

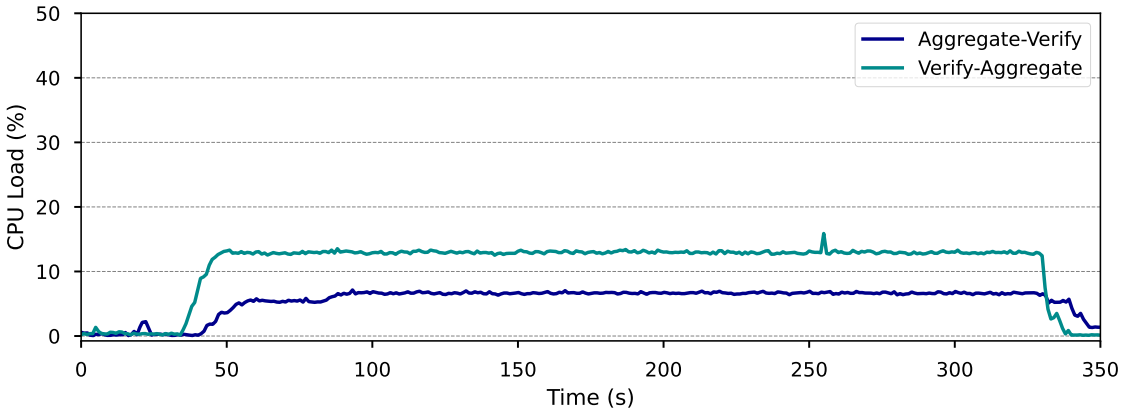

Figure 5.4: CPU Usage: global (200ms RTT - 25Mb/s links)

scenario. As expected, we see no notable difference in terms of throughput in the global scenario where the system is bottlenecking on the bandwidth and CPU. While there is a slight difference in terms of CPU load, as the system is far from being saturated, this has no notable influence on the throughput. This behavior changes substantially in the other scenarios. In both the regional and especially national scenario, the new strategy (aggregate and verify) improves considerably compared to the traditional approach. While the CPU load does not exceed 40%, this is the case as the system assumes more available resources due to hyper-threading, even though, based on our observations, when the system starts hyper-threading, there is even a performance toll involved from the additional scheduling in the context of this application.

5.2.2 Pipelining

Second, we will take a look at the effects of different pipelining levels on the throughput of Kauri. The results of this experiment are depicted in Figure 5.5 showing different throughput levels on the y-axis corresponding to different pipelining stretches on the x-axis. This experiment was executed in the geo-distributed setting (200ms RTT and 25Mb/s bandwidth) for $N = 100$. In order to fully reason about the impact of pipelining, we executed the experiment for different block sizes ranging from 50Kb (125 transactions per block) to 500kb (2000 operations per block).

We can compare the results from Figure 5.5 with the values we may obtain from our theoretical model in Table 4.3. Thus, we see that the optimal pipelining level, in this case, is very close to the value we have obtained in the theoretical model. Thus, showing that our theoretical performance model can serve as a guideline to configure our system to achieve high throughput. As a result, the experiments presented in this section orient their respective pipelining configuration to the value obtained from the theoretical performance model.

Further, we note that for all block sizes besides the smallest, eventually each configuration of Kauri, independent of the block size, will reach a similar throughput level by

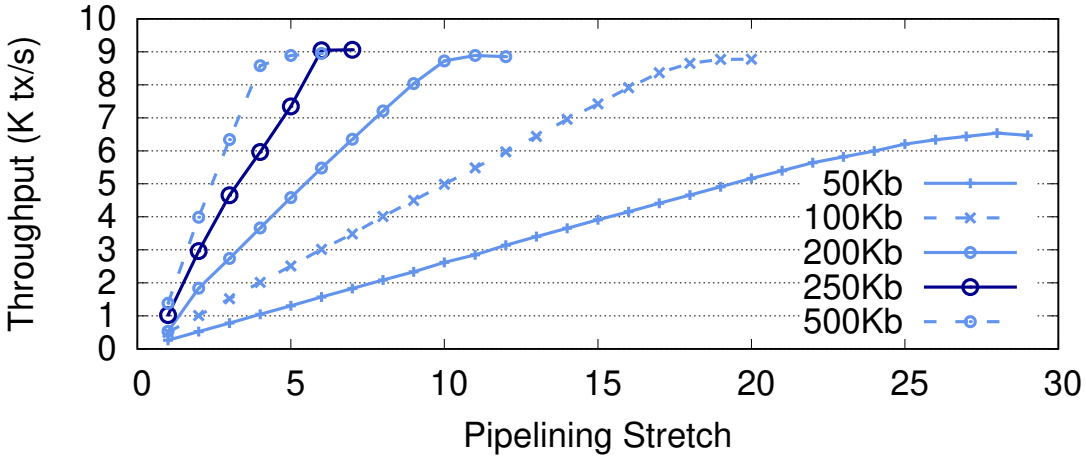

Figure 5.5: Effect of pipelining stretch on Kauri's throughput for $N = 100$, $m = 10$ and different block sizes.

compensating the smaller throughput, resulting from the smaller block-sizes, by sending blocks more regularly. The only configuration with a significantly smaller throughput is at the smallest block size. This indicates that the systems' computational resources were maxed out at this block frequency.

Finally, we can observe an almost linear performance increase for each pipelining level, which on the one hand shows the minimal overhead inherent to this approach but on the other hand, given there are enough networking and computational resources, shows we could scale the throughput further up.

As a result of this experiment, in the context of this evaluation, we picked the default block size to be 250Kb (1000 transactions per block), as this resulted in lower resource usage but still allows flexible adjustment of the pipelining level. In the original HotStuff Paper [1] the chosen size was between 100 and 400 operations per block. However, due to the lack of pipelining, HotStuff strongly benefits from the increased block size as not only will HotStuff spend less time idling, but the signature overhead makes up a significantly smaller share of each block.

5.3 Throughput

In this section, we take a look at the throughput Kauri provides, compared to other solutions in different scenarios.

5.3.1 Latency Compensation

Next, we want to show how pipelining in Kauri can, orthogonal to compensating smaller block-sizes, also allow to compensate inherent networking delays. Thus, to evaluate this, we executed an experiment with vanilla HotStuff (using secp256k1 signatures) and Kauri

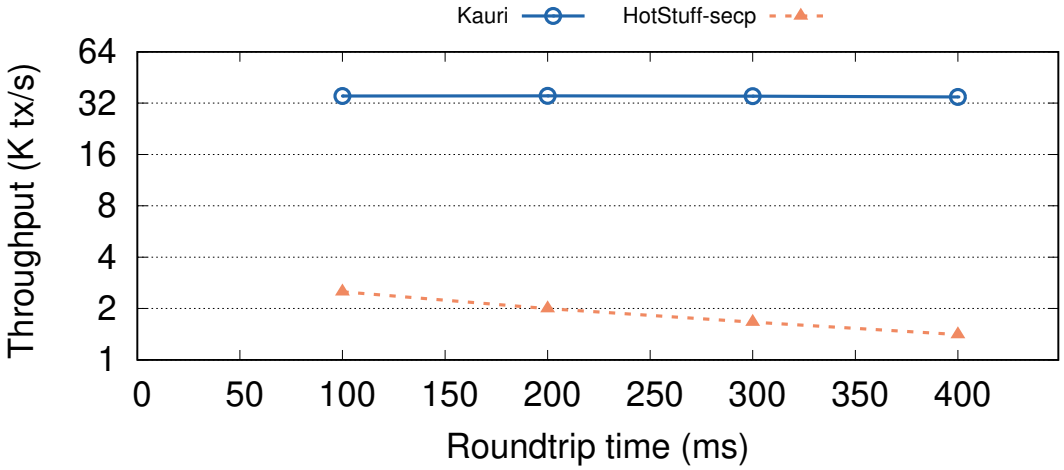

Figure 5.6: Impact of RTT in system throughput. (N=100 with 100Mb/s bandwidth)

where we have altered the inherent system latency and observed the effect on the overall system throughput.

The results of this experiment are depicted in Figure 5.6 for a system with $N = 100$ Nodes and 100Mb/s bandwidth. We fixed the bandwidth, block size, and the number of processes in the system and varied the round trip latency from 100ms to 400ms. We have varied the pipelining stretch based on our performance model, varying from 10, for 100ms to 33 for 400ms round trip time. While we can observe the rapidly declining throughput of HotStuff for higher round trip times, Kauri, displays a stable throughput independent of the inherent system latency.

Therefore, we can conclude that, as a result, Kauri can fully leverage the available computational and networking resources independent of the system latency. In contrast, existing systems like HotStuff will not be able to offer high throughput in these kinds of scenarios.

5.3.2 Increasing Number of Processes

This takes us to the most important part of this evaluation. This section evaluates the throughput of Kauri under different scenarios and system sizes. In detail, we vary the system from 100 to 800 nodes in our three given scenarios, namely National (10ms RTT and 1Gb/s Bandwidth), Regional (100ms RTT and 100Mb/s bandwidth) and Global (200ms RTT and 25Mb/s Bandwidth).

We compare the performance of Kauri with the vanilla implementation of HotStuff (HotStuff-secp), our BLS version of HotStuff (HotStuff-bls) and our implementation of Motor (Motor*[1]). The results of this experiment are depicted in Figures 5.7, 5.8 and 5.9 for the three different scenarios.

[1] We denote Motor as Motor*, as based on its implementation, it behaves exactly like Kauri without additional pipelining

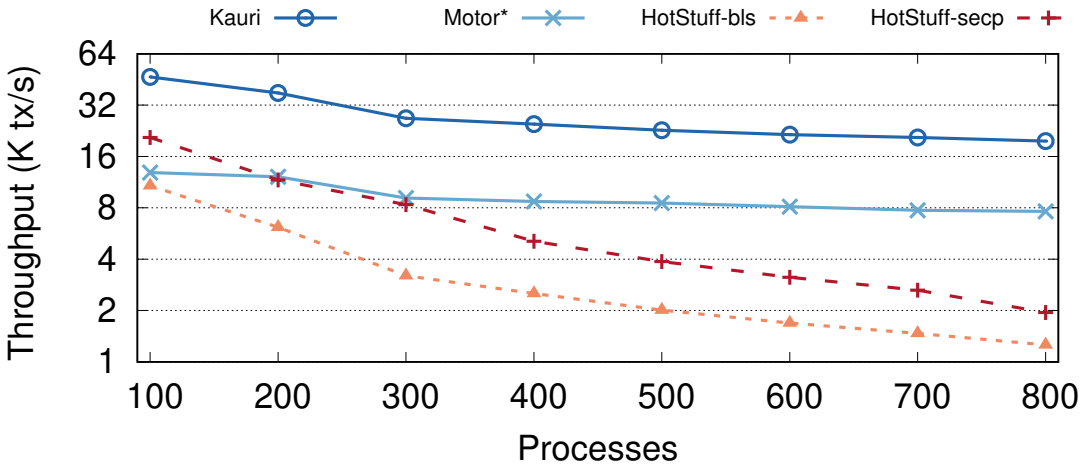

Figure 5.7: Throughput: national (10ms RTT - 1Gb/s links)

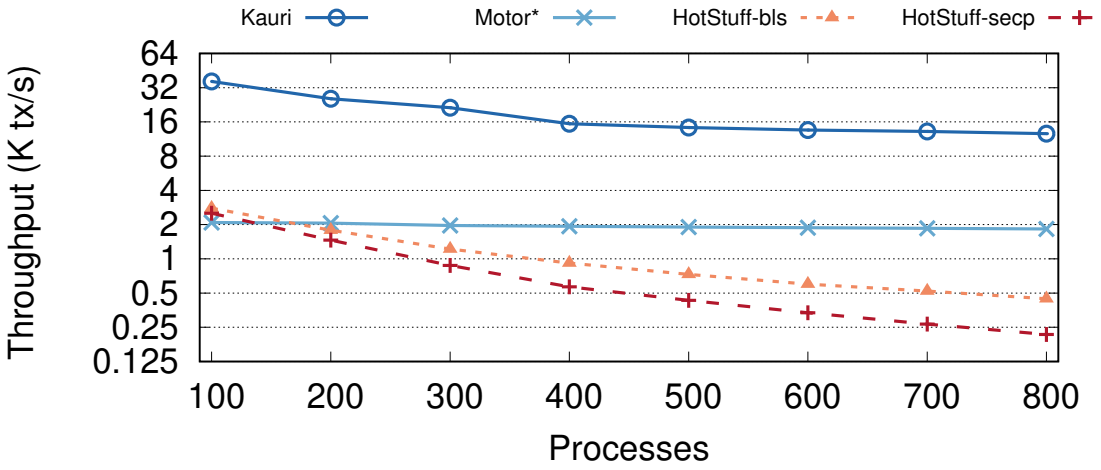

Figure 5.8: Throughput: regional (100ms RTT - 100Mb/s links)

First, very apparent, we note that Kauri outperforms any competing implementation by a large margin independent of the scenario and number of processes. This is especially apparent in scenarios with limited bandwidth and at larger system sizes. This follows the claims of our theoretical model very tightly as we have discussed in the previous section. In HotStuff, the leader process has to disseminate its block to an increasing number of nodes, consuming an increasing amount of bandwidth. Thus, as a result, the larger the set of nodes, the longer it takes for the leader to disseminate the block, and the longer it takes for the leader to start processing and disseminating the next block. On top of that, vanilla HotStuff uses secp256k1 signatures without any aggregation mechanism. As such, larger sets of signatures have to be disseminated alongside the block at larger numbers of participants, which further aggravates the problem. Thus, as a result, in all scenarios where the system is bound by the bandwidth, HotStuff-bls outperforms the vanilla implementation of HotStuff.

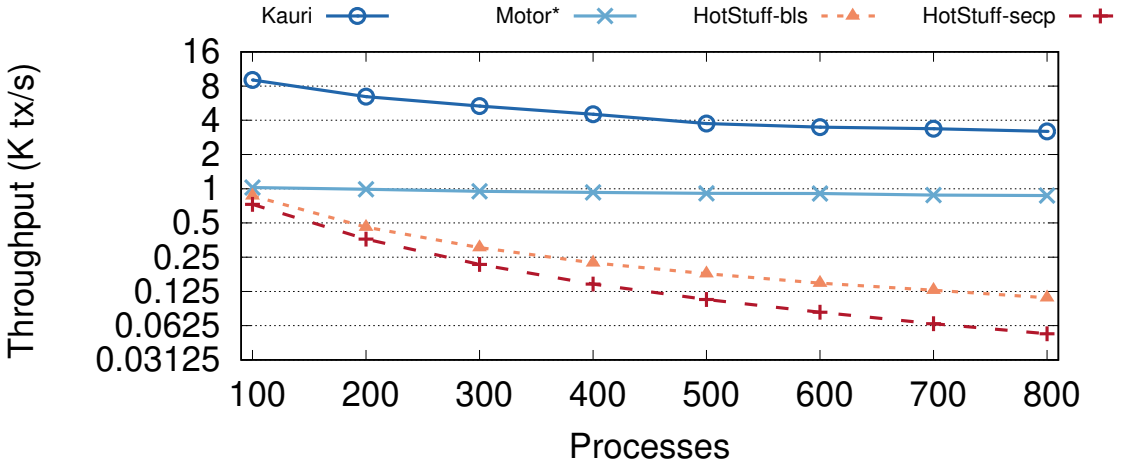

Figure 5.9: Throughput: global (200ms RTT - 25Mb/s links)

Second, we notice that while basic tree-based approaches like Motor, for smaller sets of processes, are behind HotStuff, with growing numbers of processes, tree-based approaches eventually become more efficient independent of the used scenario as trees not only allow for offloading the inherent bandwidth cost but also the inherent computational cost of verifying N signatures each round (In the global scenario for all N, in the regional scenario for $N > 200$ and in the global scenario for $N > 300$). However, tree-based approaches are heavily limited by the system latency and offer significantly less throughput than Kauri. As such, simply switching to a tree-based approach is not a silver bullet to the scalability problem. Only in combination with pipelining, Kauri can offer a considerable throughput advantage in any scenario.

Very interestingly, even though our performance model is simplistic (i.e., we assumed that computation and dissemination fully overlap), the predicated speedup values in Table 4.3 are very close to reality, reaching a speedup of over 30 for 400 nodes in the regional and global scenario. In total, the system reaches a speedup of slightly under 60 for 800 nodes in the global scenario compared to vanilla HotStuff and around 28 for our HotStuff implementation using BLS signatures which even tightly fits the strongly simplified maximum speedup of $\frac{N-1}{m}$.

We note that, throughout this experiment, we assumed a fixed depth for Kauri of $h = 2$ and adjusted the fanout m accordingly. Thus, as the system always fully operated at its limit, we fully expected a declining throughput level at larger system sizes. However, in the following section, we will show how maintaining a stable fanout allows the system to maintain a stable throughput even with an increasing number of processes.

Finally, we show how the system can maintain high throughput independent of the number of processes by maintaining a stable fanout m and adjusting the tree sizes accordingly. The results of these experiments are shown in Figure 5.10 for the global scenario and, similarly to before, from 100 up to 800 processes. In this context, there is a tree of depth two in the case of 100 processes and a tree of depth three, for the executions from

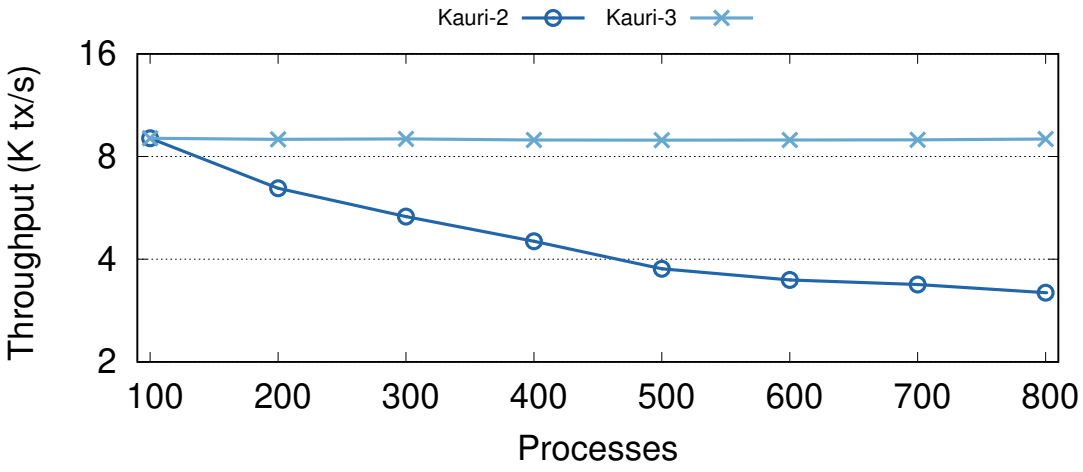

Figure 5.10: System Throughput with stable fanout

200 to 800 processes (considering the fanout $m = 10$, a tree of depth two might have at most 111 processes and a tree of depth three up to 1110 processes).

As we can clearly see in this deployment, while Kauri with a depth of two offers continuously lower throughput for larger system sizes if we increase the tree depth, we can offer a stable throughput independent of the system size.

Note, in most of the remaining experiments we consider practical system sizes of 100 processes to simplify the deployment and reduce the requirement of physical resources. We emphasize that this highly favors HotStuff as the star topology suffers strongly in scenarios with large numbers of processes due to the bandwidth and computing bottleneck.

5.4 Latency

The previous experiments showed how Kauri achieves better throughput than HotStuff independent of the system latency and bandwidth due to the load distribution in the tree and the pipelining stretch. We now conduct a study concerning the latency of our approach in comparison to HotStuff.

5.4.1 Bandwidth vs Latency

As Kauri uses a tree structure to disseminate and aggregate messages, Kauri has a higher inherent round trip latency (depending on the tree height) than HotStuff where, considering infinite computational and network resources, HotStuff will also show much lower latency than Kauri.

Nonetheless, in practice, neither bandwidth nor processing resources is infinite, and system latency is inherently bound by the sending and processing times. As such, as Kauri can reduce the sending time substantially due to the parallelism opportunities offered by the tree, in certain scenarios Kauri may even offer better latency than HotStuff.

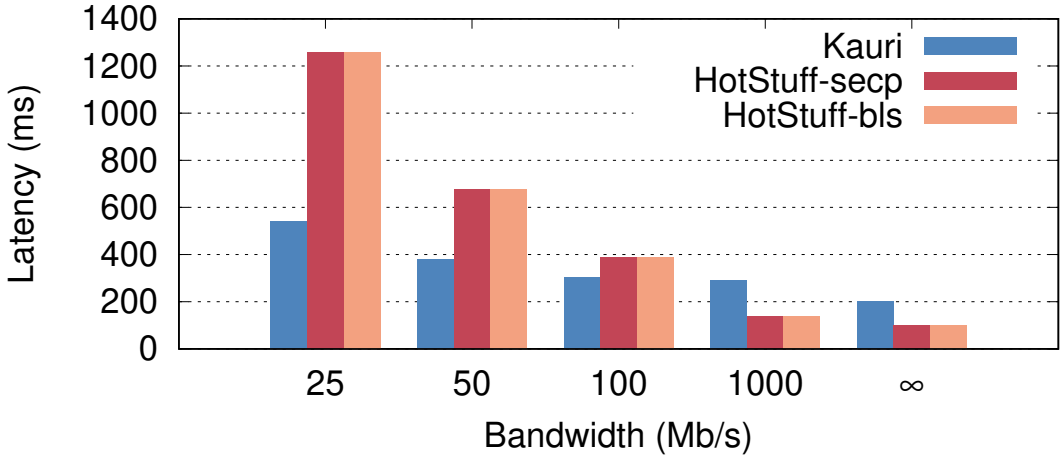

Figure 5.11: Impact of bandwidth on latency (N=100, RTT=200ms).

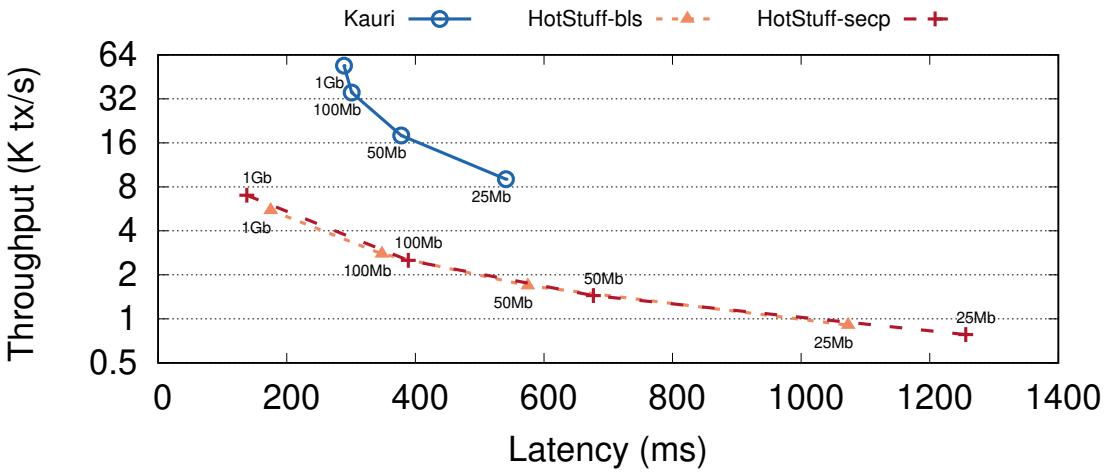

Figure 5.12: Throughput/Latency tradeoffs for different network bandwidth using trees of different heights

In order to evaluate latency, we measure the time it takes for a single quorum to be obtained for a given set of transactions (block) and, as such, do not consider potential client waiting times until the next block is produced.

In order to evaluate how this compares in practice, we executed a scenario considering a fixed RTT of 100ms and varied the bandwidth from 25Mb/s to 1000Mb/s. The results of this experiment are shown in Figure 5.11 for a system size of $N = 100$. First, we note that when sufficient bandwidth is available (i.e., 1000Mb/s), HotStuff will have approximately half the latency. However, with decreasing available bandwidth, Kauri already shows an advantage at 100Mb/s bandwidth and less than half the latency at 25Mb/s due to the significantly reduced sending time.

Figure 5.12 shows how the throughput/latency throughput compares in this scenario. Showing that the latency of both variants of HotStuff vary substantially with the available

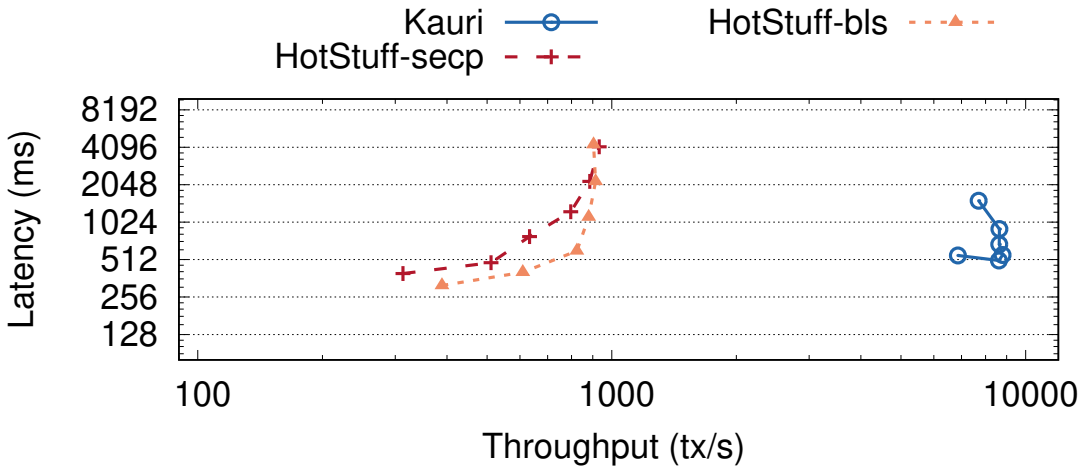

Figure 5.13: Througput-Latency for increasing block sizes (N=100, RTT=100ms, block size from 32Kb to 1Mb).

bandwidth while Kauri is much less affected. Not only do we see that HotStuff-bls may outperform HotStuff-secp in some scenarios, but that independent of the used signature scheme, both approaches have a similar characteristic in terms of the *sending time*. As such, it is crucial to significantly reduce the *sending time* if higher throughput and lower latency are required.

5.4.2 Throughput vs Latency

Next, we evaluate how the system behaves under varying load. In order to do this, we fixed the system size again at 100 processes in the global scenario (25Mb/s bandwidth and 200ms RTT). We vary the load in the system by varying the number of transactions per block (from 125 to 4000), resulting in the following block sizes: 32Kb 64Kb, 125Kb, 250Kb, 500Kb, and 1Mb.

The results of this experiment are depicted in Figure 5.13. As in all deployments of Kauri, we adjust the pipelining stretch for each scenario following our performance model. Analogous to the previous experiments, the throughput of Kauri is significantly higher than of either HotStuff variant independent of the system load.

Similarly, at lower bandwidth and increased system load, the sending time increases more substantially for HotStuff than for Kauri. As a result, Kauri offers better throughput and latency for all load scenarios above 125 transactions per block.

This further highlights the importance of using a tree to distribute both processing, but especially also the bandwidth load among a set of processes. Similar to previous experiments, HotStuff-bls shows in most scenarios a slight advantage as it does not have the additional inherent bandwidth overhead of sending a set of N signatures.

However, we have to point out a few oddities in this graph. First, we see a decrease of throughput at the highest latency level for Kauri. This stems from the fact that when the pipelining stretch is minimal, in some cases, to achieve optimal throughput, we would

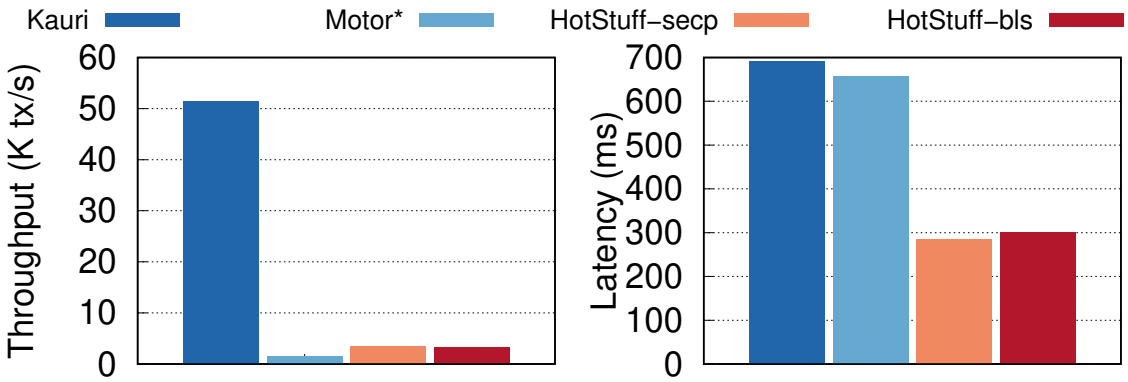

Figure 5.14: Throughput in the network of [58] with $N = 60$.

want to pipeline at most a quarter of a block or less. However, we can only increase the stretch by whole numbers, resulting in a drop in throughput.

Next, we notice a slight drop in latency at higher throughput when going from the left-most point of Kauri to the one next to it. This stems from the fact that we reached a computational bottleneck at the smallest block size, resulting in a significantly increased processing time, which, subsequently, also negatively influences the latency.

5.5 Heterogeneous Deployment

Up to this point, all experiments considered a homogeneous network where the latency and bandwidth between any two given processes are exactly the same and have analyzed how the system behaves under a range of different homogeneous scenarios.

This experiment shows how Kauri compares in a heterogeneous setup. As we lack the resources to execute the experiment in an actual globally distributed network, we have used the real-world scenario that was measured experimentally and used in the evaluation of ResilientDB [58].

5.5.1 Standard Distribution

In detail, the authors measured the latency and bandwidth between a set of google data centers in different geographic locations—namely, one each in Oregon, Iowa, Montreal, Belgium, Taiwan, and Sydney. The latency within a datacenter is roughly 0.25ms, and the latency between datacenters ranges between 38ms (Iowa to Oregon) and 270ms (Belgium to Sydney). Similarly, the bandwidth within a datacenter is roughly 10Gb/s and varies from 670Mb/s (Iowa to Oregon) and 66Mb/s (Belgium to Sydney).

We deployed 10 processes in each datacenter, resulting in a total of 60 processes matching one of the deployment scenarios in [58].

As ResilientDB is deployed statically (i.e., most of the communication happens within a cluster), we have attempted to optimize the deployment of HotStuff and Kauri. As such, the leader (root) process is always located in Oregon (best bandwidth latency to the remaining clusters), and we have distributed processes so that they are approximately close to their parent.

The results of this experiment are depicted in Figure 5.14. Similar to previous experiments, Kauri outperforms any other system in terms of throughput and, also analogous to the other deployments, this primarily stems from the use of pipelining, which allows Kauri to compensate the large round trip latencies. However, as expected, Kauri is penalized in terms of latency, as HotStuff's throughput is impaired mainly on high latency scenarios, it neither bottlenecks on bandwidth nor on the processing load. Nonetheless, Kauri displays an almost ten times higher throughput compared to only twice the latency cost.

The most interesting takeaway is that Motor* (i.e., Kauri without pipelining) exhibits the worst throughput of all approaches. This not only shows the strong effects geographic distribution has on existing tree-based approaches but especially exhibits the importance of pipelining in order to display a large throughput.

We also want to note that the values obtained for HotStuff differ substantially from those reported in ResilientDB. This stems from the fact that in the evaluation in ResilientDB [58] N parallel instances of HotStuff were run, and the reported throughput is a sum of all the N parallel instances. Contrary, we opted to consider the best possible throughput of a single instance in our deployment. Nonetheless, the throughput we have obtained for Kauri is very similar to the throughput obtained by ResilientDB, without requiring to sacrifice the resilience of the system as ResilientDB tolerates at most $f \leq \left\lfloor \frac{C-1}{3} \right\rfloor$ failures, where C is the size of the smallest cluster (§1), while Kauri tolerates $f \leq \left\lfloor \frac{N-1}{3} \right\rfloor$ faults as classical BFT.

While we manually assigned the leader to the Oregon datacenter and distributed the remaining internal nodes equally over the other data centers, this is still far from a perfect distribution as there was still regular cross-data center traffic involved.

5.5.2 Optimal Distribution

In order to achieve perfect distribution, we either have to use an irregular tree or alter the number of processes in each data center. In order to assess this, we have executed an additional experiment in the same heterogeneous deployment. Instead of having statically 10 processes in each datacenter, the first data center (Oregon) has 8 processes (1 root node, 1 internal node, 6 leaf nodes) and the remaining datacenters 7 (1 internal node, 6 leaf nods) where each internal node is solely connected to leaf nodes within its own datacenter. This resulted in a total of 43 processes.

The result of this experiment is depicted in Figure 5.15. We see a slight increase in throughput in Kauri (primarily due to the reduced fanout), reaching around 55k tx/s.

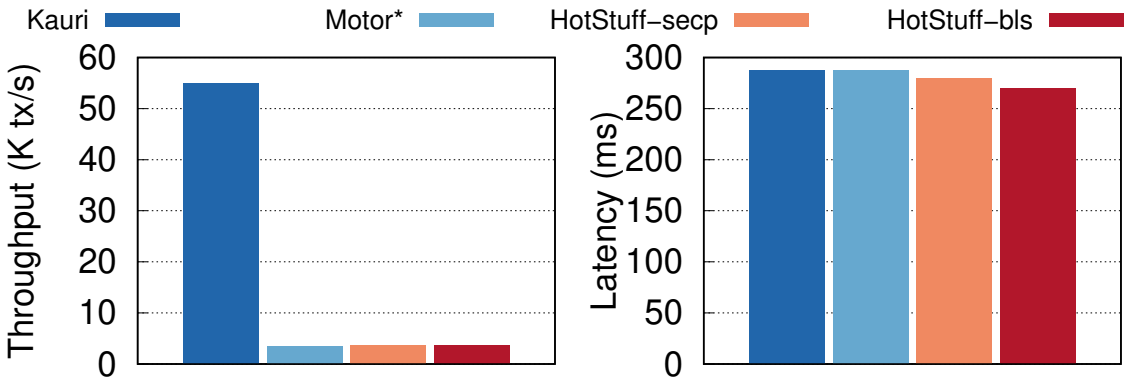

Figure 5.15: Throughput in the network of [58] with $N = 43$.

However, the main difference is the latency. Compared to almost twice the latency in the previous deployment, there is only a minimal latency overhead compared to HotStuff due to the improved deployment. This not only shows how Kauri can, when appropriately configured, show similar latency as HotStuff even outside of saturated scenarios but also strongly motivates the creation of deployment algorithms that consider geographic proximity.

5.6 Reconfiguration

Lastly, we evaluate how Kauri behaves in the presence of faulty nodes. We constructed four scenarios: 1 faulty process, 3 faulty processes, 10 faulty processes, and f faulty processes.

We have executed this experiment in the regional scenario (100ms RTT latency and 100Mb/s bandwidth) with 100 processes. As such, there are $f = 33$ faulty processes in the fourth scenario.

Our reconfiguration algorithm can always recover within optimal steps in the first three experiments by constructing disjoint sets of processes. As such, the worst-case scenario for any approach (Kauri or HotStuff) is subsequent faulty leaders. In the third experiment (10 failures), the newly designed reconfiguration algorithm comes into play and executes one additional rotation (i.e., the original algorithm tolerates up to 9 faulty processes and the refined algorithm up to 19).

However, neither algorithm can construct a robust tree in the fourth experiment, and we have to fall back to a star. This could either be after selecting f subsequent faulty leaders or, in the worst case, first 19 instances of sufficient faulty internal nodes, followed by 33 faulty leaders.

As in HotStuff, reconfiguration is triggered with a timeout. We consider an initially step timeout of $1s$ ($\delta = 0.25s$) after which a process assumes the topology to be non-robust

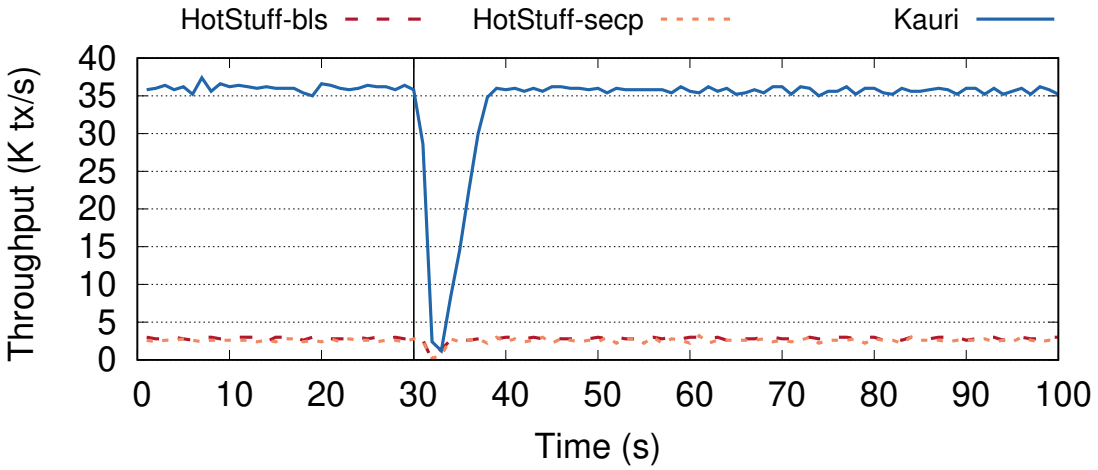

Figure 5.16: One faulty leader.

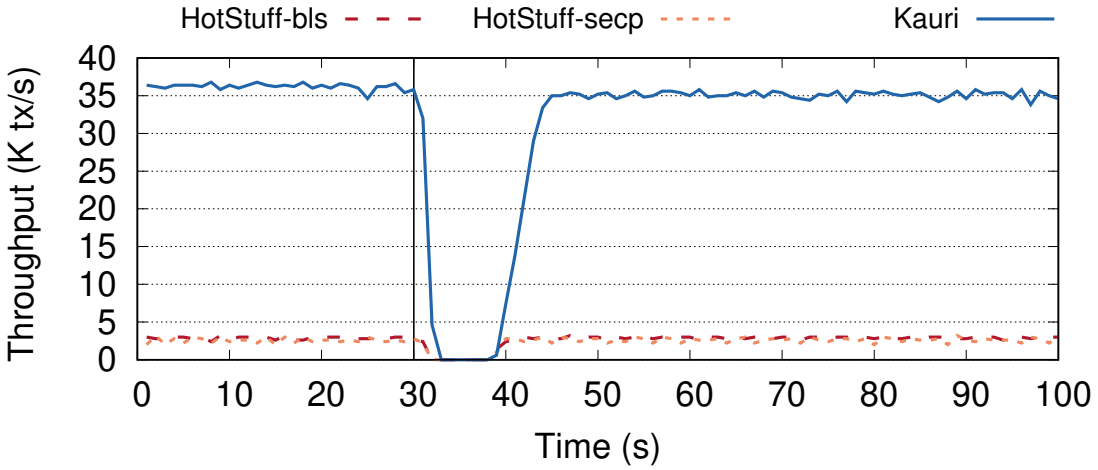

Figure 5.17: Three consecutive faulty leaders.

and triggers a reconfiguration. With subsequent failures this value doubles until reaching or exceeding $10s$ ($\Delta = 2.5s$) at which it is capped.

In each experiment, we first have a warm-up period of $30s$ (omitted from the graphs), after which we start the experiment. Following that, we inject the failure after another $30s$ (hence $60s$ after the start of the experiment) and measure the impact on the system throughput.

5.6.1 Faulty Leaders

First, we notice that for a single leader failure, both Kauri and both HotStuff variants recover very quickly to similar throughput levels as before within roughly 3s for one failure (Figure 5.16), and just under 10 seconds for three consecutive failures (Figure 5.17).

While Kauri recovers simultaneously, it takes a few additional seconds to reach the

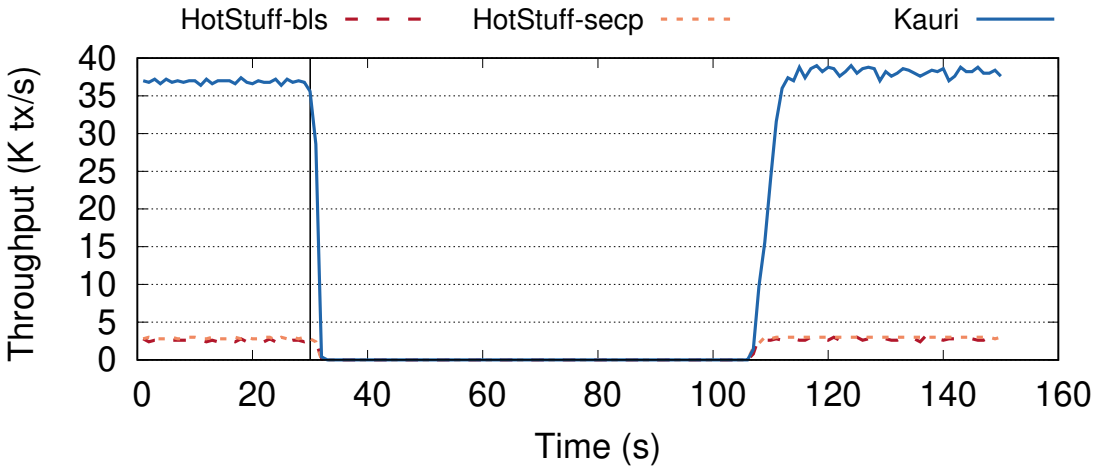

Figure 5.18: Ten consecutive faulty leaders.

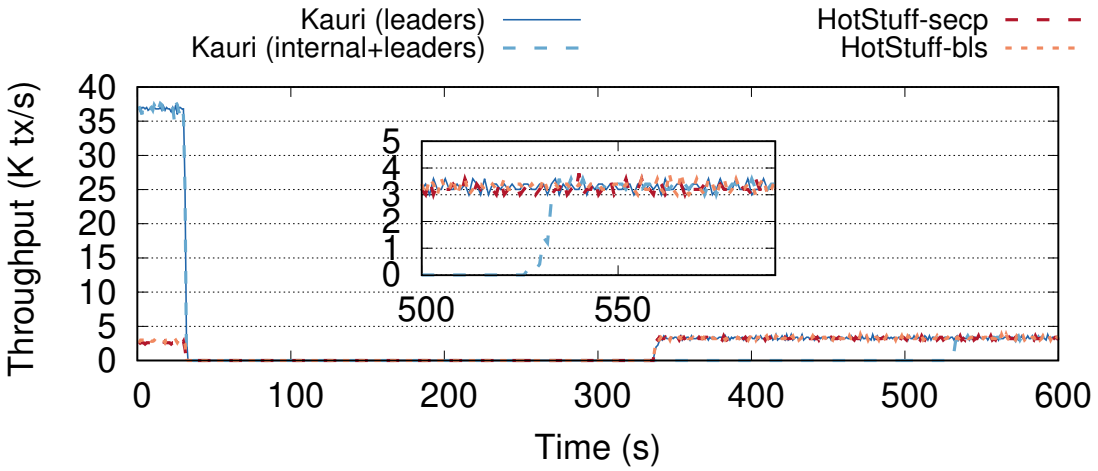

Figure 5.19: Reconfiguration at $f = \frac{N-1}{3}$

pre-failure throughput as before as Kauri has to build up the pipelining stretch again. Nonetheless, we can see that this recovery time is static, independent of the number of failures. These results are also analogous to the experiment with ten subsequent leader failures, where the extended reconfiguration algorithm we have presented was used.

5.6.2 Multiple Failure Locations

Finally, we set up a worst-case experiment with a large number of failures ($f = \frac{N-1}{3}$). As already explained, this results in two scenarios for Kauri. i) Worst case with first 19 faulty leaders in Kauri and then 14 stars with faulty leaders (labeled Kauri leaders) ii) Worst case where we got first 19 instances of faulty trees due to faulty internal nodes (labeled Kauri internal+leaders) and then 33 faulty stars with faulty leaders.

The results of this experiment are depicted in Figure 5.19. In the case of 33 consecutive faulty leaders, Kauri and HotStuff perform similarly, analogous to the previous experiments. However, due to many simultaneous failures, Kauri has to fall back to a star topology. Nonetheless, in this case, the system reacts just as quickly as HotStuff and recovers to the same performance.

Only in the absolute worst case, of 19 instances of faulty internal nodes (without overlapping the set of future leader nodes), there is an overhead of f_a rounds where we apply our algorithm. Following that, we also fall back to a star and achieve a similar performance again.

5.7 Summary

This chapter outlined the experimental results we obtained throughout the course of this thesis. It showed how Kauri compares to similar systems and validates our previously introduced performance model. The following chapter discusses the potential shortcomings of Kauri, future work and concludes this thesis.

6

Discussion & Conclusion

This chapter concludes the thesis. We start out by discussing some general considerations putting our contribution into context. Next, we discuss the lessons we have learned when designing Kauri and the requirements an approach must fulfill to scale consensus to large system sizes. After that, we discuss the limitations of our approach and in which context other protocols might be more suitable. We follow up with future directions in which Kauri can be extended in future work. Next, we discuss some orthogonal work we have executed in the context of this thesis. Finally, we conclude this thesis with a short summary and some final considerations.

6.1 General Considerations

Scaling byzantine fault tolerance consensus is a well-known problem in the literature. In Chapter 3 we outlined the most important byzantine fault-tolerant consensus algorithms as well as recent attempts to scale byzantine fault-tolerant consensus to larger system sizes while maintaining high throughput.

However, besides changes to the consensus algorithm itself, orthogonal solutions in the literature aim to improve the overall throughput at large scales. One of the most common ways is sharding, where processes are divided into groups, and each group is individually responsible for a subset of the operations. This way, through sharding, it is possible to run several rounds of consensus concurrently within each shard [37]. A global consensus involving multiple shards is only necessary if a transaction touches several shards. A relatively similar and recent approach are so-called "side-chains". We can understand side-chains like individual shards that process and batch transactions externally and regularly synchronize the results [63].

These solutions are fully compatible with our tree-based approach and pipelining scheme. Individual shards may still be geographically distributed and, as such, have a potentially low throughput due to the large idle time. In addition, the load at each individual process increases with increasing throughput, and tree structures can be used to distribute the load more equally within the shard.

Nonetheless, sharding has one thing in common with most other solutions that attempt to scale byzantine fault-tolerant consensus. Similar to approaches like DPoS or Committee-based solutions, they sacrifice resilience to increase the system's scalability where the failure assumption is usually based on the size of each shard and not on the total system size.

Outside of sharding, a recent popular approach assumes a multi-leader architecture where multiple rounds of consensus are executed in parallel [65]. This is usually done by dividing the transaction space, similar to sharding, however, without reducing the system's overall resilience. Nonetheless, these approaches are very fragile in the presence of byzantine failures, while, our tree approach tolerates faulty internal nodes at a small cost. A single faulty leader may slow down the system to 10% of its capacity in a multi-leader system [65].

Due to this, they have to guarantee somehow that none of the leaders are faulty (which is orthogonal to our strong robustness criteria). As such, they could leverage our simple reconfiguration algorithm to construct disjoint leader sets. Nonetheless, this only works at a significantly reduced normal-case failure assumption. In addition to that, it is still unclear how multi-leader approaches deal with conflicting transactions.

In comparison, Kauri offers a solution that can easily be applied to existing systems to increase their scalability with minimal trade-offs compared to most orthogonal approaches.

6.2 Lessons Learned

Next, we outline the lessons we have learned from designing Kauri and the requirements we have identified to build a scalable byzantine fault-tolerant consensus algorithm. Following that, we also discuss precautions that must be taken into account when working with tree structures.

6.2.1 Scalable Consensus

The three main inhibitors of scaling consensus are pretty straightforward. First, the protocol can run into a bandwidth bottleneck depending on the number of messages and data transmitted. Second, with an increasing set of processes, more computational power is required to verify the messages of all processes. In addition to that, as we've shown in the evaluation, high network latency reduces the potential throughput significantly unless an optimistic pipelining scheme is used. However, balancing bandwidth and computational load among a set of processes by using deeper communication structures leads to more communication steps, and as such, the higher round trip latency in a growing system impacts the throughput even more significantly.

6.2.1.1 Bandwidth

Most modern consensus algorithms are leader-driven. In practice, a single process proposes a set of values (transactions) by propagating it to all other participants. Assuming 100 consensus participants and a batch size (block size) of 1Mb, this consumes 100Mb/s of bandwidth per round of consensus. As such, quite clearly, the bandwidth consumption scales linearly with the number of participants and eventually leads to a bottleneck. Even if the leader process rotates after each round of consensus, the time each round takes is still essentially bound by the available bandwidth at the leader. This is even further aggravated by the requirement to propagate the set of signatures alongside.

The signature cost can be reduced through a signature aggregation scheme as in SBFT [27]. Meanwhile, the best way to reduce the bandwidth cost of propagating the initial proposal is by adapting the communication structure. Thus, instead of broadcasting the set of transactions to all processes, the set can be propagated through a tree. As a result, instead of requiring N messages at the leader, each process only has to propagate at most $fanout = m$ messages. As such, it is either possible to maintain a stable fanout (at the cost of increasingly deep trees) or adjust the fanout to maintain a tree of fixed size (at the cost of throughput). Nonetheless, even for a fixed size tree to preserve latency, as we have shown in the evaluation, m always grows significantly slower than N.

While it is also possible to attempt this through gossip, byzantine fault-tolerant gossip requires significantly more messages and is not as flexible as a tree structure which can be adapted depending on the available bandwidth at each node [42].

6.2.1.2 Computational Complexity

In terms of computation, in most algorithms, at least one process has to verify the signatures of all participants. This again is a very obvious bottleneck as growing system sizes eventually require an increasing computational cost at this process. Independent of the used cryptographic scheme, the inherent cost at the leader is always $\mathcal{O}(N)$.

Again, the best way to alleviate this, is through a tree structure, where inverse to propagation, signatures are aggregated on the way back to the leader at $\mathcal{O}(m)$ complexity at each internal node. Therefore, adjusting the fanout m accordingly to the latency and throughput requirements is again possible.

Similar to before, aggregation through gossip as in [42] is also possible, but by far, it is not as efficient as a tree structure.

6.2.1.3 Latency

While both computation power and bandwidth can be distributed among multiple nodes by using hierarchical communication structures, this introduces additional communication steps in the system. This is especially complicated in a geo-distributed system

where additional communication steps can significantly increase the consensus latency, resulting in a severe drop in throughput.

As such, it is essential to decouple throughput from the system latency. HotStuff leveraged pipelining to compensate for the additional consensus phases. However, this neither accounts for latency introduced by geographic distribution nor does it account for deeper communication structures. As such, there is a need to process several consensus rounds in parallel in the system. A straightforward solution for this is the proposed additional pipelining stretch, where the leader process proposes several blocks optimistically in a row.

6.2.1.4 Summary

The proposed approach in this thesis works due to the unique combination of several elements. Signature aggregation to reduce computation and bandwidth cost, usage of a tree structure for load distribution, and extra pipelining to compensate for the additional inherent tree latency. This becomes clear when looking at approaches in the literature that leveraged only a subset of these strategies and were unable to achieve a comparable throughput (more details in Section 3 and Section 5).

6.2.2 Disadvantages of Trees

Even though tree structures are an excellent solution to distribute the load among a set of processes, in the context of byzantine fault tolerance, they are significantly more fragile as a faulty (non-leader) node in a tree structure has significantly more power than a faulty "leaf" node in a clique or star. Increasingly deep tree structures require more internal nodes, which results in a more significant chance to encounter faulty processes among the internal nodes in the tree.

As such, digital signatures are essential to reduce the potential attack vectors significantly. If each message is authenticated through a unique key-pair, the worst attack a faulty internal node can execute is failing to relay signatures from/to their child nodes. However, sufficient faulty internal nodes may still block consensus from terminating indefinitely.

Thus, contrary to star and clique schemes, we have to reconfigure the system, not only in the presence of a faulty leader but also if there are too many faulty internal nodes. Thus, intelligent reconfiguration algorithms are necessary to avoid a factorial number of steps until a robust tree is found and consensus can be achieved.

These algorithms are not only interesting for tree construction but, as discussed, in the context of multi-leader schemes, can be used to construct a robust leadership as well.

6.3 Limitations

While Kauri offers high performance at a large scale, our approach still has certain limitations. We identify three main cases where other approaches might be more suitable than Kauri.

6.3.1 Datacenter Environments

Kauri was designed for high throughput environments where either computational power or the available bandwidth is limited. Thus, if all system processes are situated in the same data center, it is unlikely that the system will bottleneck on bandwidth. Meanwhile, the computational cost mainly stems from the usage of asymmetric signatures. Thus, as bandwidth is not a limiting factor, an all-to-all scheme could be used that relies on message authentication codes (MACs) to reduce the computational cost.

On top of that, in a data center environment, timing assumptions are different compared to geographically distributed environments, and, as such, consensus protocols that rely on synchronous communication become a suitable option [57]. Finally, one could argue that datacenter environments allow relaxing the failure assumptions, and committee-based solutions also become an appealing alternative.

6.3.2 Small Consensus Groups

While Kauri outperforms HotStuff already in a system with 100 processes, Kauri was designed for large scale consensus. As such, small systems that do not require 100s of processes but numbers in the single or double digits can easily rely on all-to-all consensus schemes like PBFT [16]. Not only are these approaches slightly more straightforward than Kauri, but they have shown numerous times to offer high throughput in these environments.

6.3.3 Small Application Latency

While Kauri offers excellent throughput, the client-side latency is significantly higher than in other approaches due to the use of a tree. It requires more communication steps per round. In situations where the system is saturated, the tree approach can, as we have shown, display better latency than other approaches. However, during normal case operation, the latency might be significantly higher. As such, Kauri is not suitable for applications that require short response times for clients (e.g., real-time applications, games, etc.). In these cases, it might be advantageous to use approaches that create regional shards like ResilientDB [58] that can approve most client transactions within the regional shard resulting in very low client-side latency. Nonetheless, we stress that if local shards are sufficiently large, combining Kauri with a sharding-based approach is possible to increase the shard's throughput.

6.4 Future Work

While we pointed out certain limitations in the previous section, we have left out the cases we envision as future work. As such, while Kauri still lacks certain features that might be important in some use-cases, we believe that natural extensions of Kauri could fulfill these requirements.

6.4.1 Current Work in Progress

Outside of the already published papers, we are currently finishing a paper to submit to ACM talks, including the latest experimental results and the extended reconfiguration algorithm.

6.4.2 Dynamic Pipelining

While in the context of this thesis, we assumed pipelining to be statically pre-configured, in practice, we expect system parameters to change during runtime, and, as such, the pipelining stretch has to adapt to that as well. Thus, an important extension of Kauri would involve developing algorithms that allow the system to detect the potential to pipeline additional blocks or, inversely, detect that the current pipelining stretch is oversaturating the system.

While this seems like a straightforward extension, it still requires careful consideration, as we assume geographic distribution, a partial synchronous environment, and, on top of that, byzantine failures (e.g., faulty processes could slow down the system just enough to trick the system into increasing or decreasing the pipelining level).

6.4.3 Locational Awareness

In our evaluation, in most cases, we configured a homogenous network where all processes have the same bandwidth/CPU, and the latency between any two processes is constant.

However, in practice, in a geo-distributed blockchain environment, the reality is a very heterogeneous network with broadly different latency and bandwidth distributions. Thus, a reconfiguration algorithm that takes locational data and available bandwidth into account can not only significantly impact the system throughput but especially also the clientside latency.

As we have shown in the evaluation, if trees are configured in a way such that geographically close processes also communicate with each other in the tree, it is possible to reduce the latency significantly. For example, if the latency from the root to the internal nodes is 100ms, but from internal to leaf nodes, it only takes 5ms, the latency trade-off compared to a star becomes unsubstantial. Thus, in the best case, the latency of a tree could be very similar to a star.

Nonetheless, designing these algorithms is not trivial. Not only do we have to assume that byzantine processes will try to influence the system in their favor, but also if certain processes are more likely to be internal or root nodes, this could also influence the robustness of the tree structure.

6.4.4 Regular Tree Rotation

In Kauri, we assume a static leader that only changes upon failure or timeout. However, a faulty leader could, e.g., censor certain client transactions without being detected by the remaining processes. This is especially a problem in the blockchain setting, where specific nodes might be unable to accept subgroups of requests due to government regulations or sanctions. In addition to that, in the presence of faulty internal nodes, correct leaf processes might not receive the consensus results for long periods of time.

Thus, regularly reconfiguring the tree yields certain benefits. However, this also comes with certain disadvantages. First, it is more likely to find non-robust trees, and as such, there are more reconfiguration periods where throughput is strongly affected. Second, it is harder to pipeline effectively, as the leader is only in this position for a limited amount of time, negatively affecting throughput.

Due to these factors, specific changes are necessary. The reconfiguration algorithm has to be adjusted to make sure that adversaries do not know ahead of time details about the upcoming tree structures. At the same time, we want also want to make sure that the next leader is one of the current internal nodes to improve the pipelining efficiency (e.g., if the next leader is a leaf node, it takes longer for it to receive the outstanding pipelined leader proposals before being able to propose their own block).

When rotating trees regularly, it might also make sense to adapt the reconfiguration algorithm further to construct disjoint trees (in a sequence of trees, nodes are never connected to the same parent process twice). This not only reduces the chances of a correct process being connected to faulty processes for a long time period but also reduces the load on the link between two given nodes.

6.4.5 BLS Improvements

The current implementations of BLS signatures are yet very immature and are not used in many major projects in production. As such, a series of improvements could be made to these implementations to reduce the computational load. As a result, this could result in a significant throughput improvement for Kauri.

In the process of this thesis, as mentioned, we already incorporated one change in the used BLS library which resulted in an aggregation speedup by a factor of 20. However, there are still several possibilities open to improving the performance. For once, BLS signatures rely on cryptographic pairings. Due to this, the vast majority of the computational load is related to constructing a large set of pairings. Thus, a possible improvement

could be attempting to cache pairings for longer time periods to reduce the load at verification/aggregation time.

Another possibility is improved multi-threading. At the moment, signatures are processed on the same thread. Thus, a possible improvement could consist of multi-threading the verification—aggregation and even signing to reduce the system load of the algorithm.

6.5 Additional Research

Besides the before mentioned aspects about Kauri, a set of orthogonal research projects were executed during the course of this thesis, which resulted in several scientific publications. All of these contributions are related to decentralized applications that can be built on the blockchain with the help of smart contracts.

In general, we highlight three main projects.

6.5.1 HRM Smart Contracts

Our first project focused on the decentralization of human resource management. There are numerous corruption and nepotism scandals in the context of hiring procedures in public agencies due to favorable working conditions in Brazil. As such, designing this process more transparent and accountable as well as reducing the influence of local officials could re-instantiate trust into this process. We have created two smart contracts that serve as the basis of the application process. First, there is an institution-list contract where public institutions register the institution and sign-up potential reviewers. Before publishing a vacancy, institutions should apply to join the institution list contract. To do so, institutions have to prove their authenticity to the other institutions on the list. Following that, a majority vote of acceptance of the existing institutions is required. After an institution has signed up, they can register potential reviewers at their own discretion.

The other type of smart contract is the vacancy contract. If an institution wants to publish a vacancy, they create a smart contract with matching parameters and publish it on the blockchain. Following that, applicants can send their application data to the corresponding institution and register with the smart contract, including a hash of the application data and a small security deposit. After the initial application phase is over, the smart contract stops accepting new applicants and invokes the institution list contract to assign the application process to a random set of reviewers from different institutions. Reviewers query the application data from the institution (to avoid publishing personal data publicly on the blockchain) and post the decision in the smart contract (to remain anonymous, reviewers can prove their selection as a reviewer by signing a message with their private key). In each phase, the reviewers rank the applicant based on the criteria of the given phase. The smart contract sorts out the reviews that differ too much from the average and rewards reviewers that came to a similar conclusion with a share of the application fee. If applicants send fraudulent data, reviewers can also report that, and

the applicant loses their security deposit (to avoid incentivizing misreporting this, the security deposit gets "burned").

We have implemented the smart contract on two different blockchains; On Ethereum using solidity as a native smart contract on Ethereum and evaluated the price the execution would cost. The other implementation was in a layer-2 application that just uses the blockchain to store the state. As such, third parties can verify the smart-contract execution to detect misbehavior. This is possible in this specific use-case as we are dealing with public institutions. As such, we can assume a level of accountability, as public institutions can easily be processed if they do not follow the established protocol. Our results show that it is viable to implement many applications as layer-2 solutions resulting in a significantly reduced execution cost and decreasing the main-chain load.

6.5.2 Dagora Market

I participated in this project as the Advisor of the Bachelor Thesis of a Student. In the context of this work, we have developed a framework to create a decentralized marketplace for physical goods on the Polygon blockchain. The smart contracts were again developed in solidity, and the deployment cost on both Ethereum and Polygon was analyzed. We have evaluated the difficulties of deploying smart contracts on Ethereum side-chains and elaborated an in-depth cost comparison. Our results show a significantly cost difference between the two possible deployments at only a slightly increase in complexity for the user which could be hidden with the help of good user interfaces.

6.5.3 Side-Chain Evaluation

In this final project, we have evaluated the state of layer-2 solutions on the Ethereum blockchain. We have collected information about a range of different layer-2 projects and compared them regarding their scalability, security, and features. Furthermore, we analyzed their detailed impact on the main chain (Ethereum) and evaluated the potential maximum throughput the layer-2 project can contribute to the Ethereum ecosystem before reaching the scalability limit of the main chain.

6.6 Final Considerations

In the context of this thesis, we have intensively studied the obstacles that have to be taken into account when designing scalable byzantine fault-tolerant consensus algorithms in geo-distributed settings. We identified three main factors: the bandwidth and computational complexity and long idle times in geo-distributed settings.

In order to solve this, we have developed Kauri, a unique combination of tree-based communication, optimistic pipelining, and signature aggregation. In combination with aggregated signatures, tree-based communication distributes bandwidth equally among a set of processes. Combining this with optimistic block proposals compensated for the

inherent latency from constructing a communication tree in a geo-distributed environment.

We have extensively evaluated Kauri, achieving high throughput at any scale and latency, outperforming state-of-the-art protocols in any scenario, offering up to almost 60 times the throughput of competing approaches.

While Kauri already shows excellent performance in many scenarios, we not only bring a new solution to the table to be considered in this context but also open up a range of future research that can be executed to extend Kauri further.

Bibliography

[1] I. Abraham, G. Gueta, and D. Malkhi. "Hot-Stuff the Linear, Optimal-Resilience, One-Message BFT Devil". In: *CoRR* abs/1803.05069 (2018). arXiv: 1803.05069. URL: http://arxiv.org/abs/1803.05069 (cit. on pp. 2–4, 9, 10, 26, 31, 34, 35, 40, 71).

[2] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. "Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks". In: *IEEE Transactions on Dependable and Secure Computing* 7.1 (2010), pp. 80–93 (cit. on pp. 2, 29, 31).

[3] Z. Amsden et al. *The Diem Blockchain*. 2020. URL: https://developers.diem.com/docs/technical-papers/the-diem-blockchain-paper/ (visited on 01/01/2022) (cit. on p. 2).

[4] B. Artur, Y. M. Ermol'ev, and Y. M. Kaniovskii. "A Generalized Urn Problem and its Applications". In: *Cybernetics* 19.1 (1983), pp. 61–71 (cit. on p. 55).

[5] A. Avizienis. "The N-Version Approach to Fault-Tolerant Software". In: *IEEE Transactions on Software Engineering* SE-11.12 (1985), pp. 1491–1501 (cit. on p. 14).

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33 (cit. on pp. 12, 14, 15).

[7] A. Baliga. "Understanding Blockchain Consensus Models". In: *Persistent* 4 (2017), pp. 1–14. URL: https://www.persistent.com/wp-content/uploads/2017/04/WP-Understanding-Blockchain-Consensus-Models.pdf (visited on 01/01/2022) (cit. on pp. 28, 31).

[8] D. Balouek, A. Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Ed. by I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20 (cit. on p. 67).

[9] Block'tivity Team. "Block'tivity". In: (2022). URL: https://blocktivity.info/ (visited on 01/01/2022) (cit. on p. 2).

[10] D. Boneh, S. Gorbunov, R. S. Wahby, H. Wee, and Z. Zhang. *BLS Signatures*. Internet-Draft draft-irtf-cfrg-bls-signature-04. Work in Progress. Internet Engineering Task Force, Sept. 2020. 30 pp. URL: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-bls-signature-04 (cit. on p. 10).

[11] D. Boneh, B. Lynn, and H. Shacham. "Short signatures from the Weil pairing". In: *Journal of cryptology* 17.4 (2004), pp. 297–319 (cit. on pp. 10, 38, 62).

[12] M. Bravo, L. Rodrigues, R. Neiheiser, and L. Rech. "Policy-Based Adaptation of a Byzantine Fault Tolerant Distributed Graph Database". In: *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. 2018, pp. 61–71 (cit. on p. 6).

[13] R. G. Brown, J. Carlyle, I. Grigg, and M. Hearn. "Corda: an Introduction". In: *R3 CEV, August* 1 (2016), p. 15. URL: https://blockchainlab.com/pdf/corda-introductory-whitepaper-final.pdf (visited on 01/01/2022) (cit. on p. 2).

[14] E. Buchman. "Tendermint: Byzantine Fault Tolerance in the Age of Blockchains". PhD thesis. 2016 (cit. on pp. 1, 19).

[15] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. 2nd. Springer, 2011 (cit. on pp. 16, 34).

[16] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 173–186 (cit. on pp. 3, 4, 21, 22, 25, 31, 89).

[17] M. Castro and B. Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery". In: *ACM Transactions on Computer Systems* 20.4 (Nov. 2002), pp. 398–461 (cit. on p. 9).

[18] L. Chen, L. Xu, N. Shah, Z. Gao, Y. Lu, and W. Shi. "On Security Analysis of Proof-of-Elapsed-Time (poet)". In: *International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer. 2017, pp. 282–297 (cit. on p. 20).

[19] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'09. Boston, Massachusetts: USENIX Association, 2009, pp. 153–168 (cit. on p. 25).

[20] B. Cohen and K. Pietrzak. *The chia network blockchain*. 2020. URL: https://www.chia.net/assets/ChiaGreenPaper.pdf (visited on 11/12/2020) (cit. on pp. 11, 62).

[21] F. Cristian. "Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems". In: *Distributed Computing* 4.4 (1991), pp. 175–187 (cit. on p. 13).

[22] Y. Desmedt and Y. Frankel. "Threshold Cryptosystems". In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by G. Brassard. New York, NY: Springer New York, 1990, pp. 307–315 (cit. on p. 10).

[23] C. Dwork, N. Lynch, and L. Stockmeyer. "Consensus in the Presence of Partial Synchrony". In: *Journal of the ACM* 35.2 (Apr. 1988), pp. 288–323 (cit. on pp. 21, 34).

[24] M. Eischer and T. Distler. "Latency-Aware Leader Selection for Geo-Replicated Byzantine Fault-Tolerant Systems". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2018, pp. 140–145 (cit. on p. 24).

[25] M. J. Fischer, N. A. Lynch, and M. S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *Journal ACM* 32.2 (Apr. 1985), pp. 374–382 (cit. on pp. 21, 34).

[26] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. "Algorand: Scaling Byzantine Agreements for Cryptocurrencies". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 51–68 (cit. on pp. 2, 28, 31, 61).

[27] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu. "SBFT: A Scalable and Decentralized Trust Infrastructure". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2019, pp. 568–580 (cit. on pp. 2, 23, 87).

[28] O. Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge University Press, 2009 (cit. on p. 9).

[29] P. Gouveia, J. Neves, C. Segarra, L. Liechti, S. Issa, V. Schiavoni, and M. Matos. "Kollaps: Decentralized and Dynamic Topology Emulation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020 (cit. on p. 68).

[30] I. Grigg. "EOS, An Introduction". In: (2017). URL: https://iang.org/papers/EOS_Una_Introduccio%5C%CC%5C%81n-IanGrigg-Trad_MAM+AIH-20180717.pdf (visited on 01/01/2022) (cit. on pp. 2, 20).

[31] V. Hadzilacos and S. Toueg. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Tech. rep. USA, 1994 (cit. on p. 15).

[32] S. Hemminger. "Network Emulation with NetEm". In: *Proceedings of the 6th Australia's National Linux Conference (LCA2005)*. Canberra, Australia, Apr. 2005 (cit. on p. 67).

[33] A. Kiayias, A. Russell, B. David, and R. Oliynykov. "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol". In: *Advances in Cryptology – CRYPTO 2017*. Ed. by J. Katz and H. Shacham. Cham: Springer International Publishing, 2017, pp. 357–388 (cit. on p. 25).

[34] S. King and S. Nadal. "PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake". In: *self-published paper, August* 19 (2012). URL: https://www.chainwhy.vip/upload/default/20180619/126a057fef926dc286accb372da46955.pdf (visited on 01/01/2022) (cit. on p. 19).

[35] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. "Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing". In: *USENIX Security*. Austin (TX), USA, 2016, pp. 279–296 (cit. on pp. 2, 3, 30, 31, 42).

[36] E. Kokoris-Kogias. "Robust and Scalable Consensus for Sharded Distributed Ledgers." In: *IACR Cryptology ePrint Archive* 2019 (2019), p. 676 (cit. on pp. 3, 30, 61, 68).

[37] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding". In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 583–598 (cit. on pp. 2, 3, 30, 31, 42, 85).

[38] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. "Zyzzyva: Speculative Byzantine Fault Tolerance". In: *ACM Transactions on Computer Systems* 27.4 (2010), 7:1–7:39 (cit. on p. 24).

[39] L. Lamport. "Proving the Correctness of Multiprocess Programs". In: *IEEE Transactions on Software Engineering* 2 (1977), pp. 125–143 (cit. on p. 12).

[40] L. Lamport, R. Shostak, and M. Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401 (cit. on pp. 1, 21).

[41] D. Larimer, N. Scott, V. Zavgorodnev, B. Johnson, J. Calfee, and M. Vandeberg. "Steem: An Incentivized, Blockchain-Based Social Media Platform". In: *March. Self-published* (2016). URL: https://steem.com/SteemWhitePaper.pdf (visited on 01/01/2022) (cit. on pp. 2, 20).

[42] P. Li, G. Wang, X. Chen, F. Long, and W. Xu. "Gosig: A Scalable and High-Performance Byzantine Consensus for Consortium Blockchains". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 223–237 (cit. on pp. 27, 87).

[43] W. Li, C. Feng, L. Zhang, H. Xu, B. Cao, and M. Imran. "A Scalable Multi-Layer PBFT Consensus for Blockchain". In: *IEEE Transactions on Parallel and Distributed Systems* 32.5 (2021), pp. 1146–1160 (cit. on p. 31).

[44] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. "The Honey Badger of BFT Protocols". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 31–42 (cit. on p. 21).

[45] M. Milutinovic, W. He, H. Wu, and M. Kanwal. "Proof of Luck: An Efficient Blockchain Consensus Protocol". In: *Proceedings of the 1st Workshop on System Software for Trusted Execution*. SysTEX '16. Trento, Italy: ACM, 2016, 2:1–2:6 (cit. on p. 20).

[46] J.-W. Moon. "On level numbers of t-ary trees". In: *SIAM Journal on Algebraic Discrete Methods* 4.1 (1983), pp. 8–13 (cit. on p. 44).

[47] S. Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (2008) (cit. on pp. 1, 10, 16–18).

[48] R. Neiheiser, G. Inácio, L. Rech, and J. Fraga. "HRM Smart Contracts on the Blockchain". In: *2019 IEEE Symposium on Computers and Communications (ISCC)*. 2019 (cit. on p. 6).

[49] R. Neiheiser, G. Inácio, L. Rech, and J. Fraga. "HRM Smart Contracts on the Blockchain: Emulated vs Native". In: *Cluster Computing* (2020) (cit. on p. 6).

[50] R. Neiheiser, M. Matos, and L. Rodrigues. "Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 35–48 (cit. on p. 33).

[51] R. Neiheiser, D. Presser, L. Rech, M. Bravo, L. Rodrigues, and M. Correia. "Fireplug: Flexible and Robust N-Version Geo-Replication of Graph Databases". In: *2018 International Conference on Information Networking (ICOIN)*. 2018, pp. 110–115. DOI: 10.1109/ICOIN.2018.8343095 (cit. on pp. 2, 29, 31).

[52] R. Neiheiser, L. Rech, M. Bravo, L. Rodrigues, and M. Correia. "Fireplug: Efficient and Robust Geo-Replication of Graph Databases". In: *IEEE Transactions on Parallel and Distributed Systems* 31.8 (2020), pp. 1942–1953 (cit. on p. 6).

[53] R. Neiheiser, L. Rech, and J. da Silva Fraga. "Constantino: Uma Arquitetura BFT Escalável e Eficiente para Blockchains". In: *Anais do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuidos*. SBC. 2019, pp. 127–140 (cit. on p. 5).

[54] R. Neiheiser, L. Rodrigues, and M. Matos. "Kauri: Scalable BFT Consensus with PipelinedTree-Based Dissemination and Aggregation". In: *Proceedings of the 28th ACM Symposium on Operating Systems Principles*. SOSP '21. Online: Association for Computing Machinery, 2021 (cit. on p. 5).

[55] W. L. Nicholson. "On the Normal Approximation to the Hypergeometric Distribution". In: *The Annals of Mathematical Statistics* 27.2 (1956), pp. 471–483. URL: http://www.jstor.org/stable/2237005 (cit. on p. 56).

[56] M. Pilkington. "11 Blockchain technology: principles and applications". In: *Research Handbook on Digital Transformations* (2016), p. 225 (cit. on p. 1).

[57] D. Porto, J. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues. "Visigoth Fault Tolerance". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: Association for Computing Machinery, 2015 (cit. on p. 89).

[58] S. Rahnama, S. Gupta, T. M. Qadah, J. Hellings, and M. Sadoghi. "Scalable, Resilient, and Configurable Permissioned Blockchain Fabric". In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), pp. 2893–2896 (cit. on pp. 2, 29, 31, 33, 67, 78–80, 89).

[59] P. Rogaway and T. Shrimpton. "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance". In: *Fast Software Encryption*. Ed. by B. Roy and W. Meier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 371–388 (cit. on p. 8).

[60] F. B. Schneider. "Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Computing Surveys* 22.4 (1990), pp. 299–319 (cit. on p. 15).

[61] C. P. Schnorr. "Efficient Identification and Signatures for Smart Cards". In: *Advances in Cryptology — CRYPTO' 89 Proceedings*. Ed. by G. Brassard. New York, NY: Springer New York, 1990, pp. 239–252 (cit. on p. 10).

[62] F. Schuh and D. Larimer. *Bitshares 2.0: General Overview*. 2017. URL: https://cryptorating.eu/whitepapers/BitShares/bitshares-general.pdf (visited on 01/01/2022) (cit. on p. 20).

[63] C. Sguanci, R. Spatafora, and A. M. Vergani. *Layer 2 Blockchain Scaling: a Survey*. 2021. arXiv: 2107.10881 [cs.DC] (cit. on p. 85).

[64] J. Sousa and A. Bessani. "Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines". In: *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. 2015, pp. 146–155 (cit. on p. 24).

[65] C. Stathakopoulou, T. David, and M. Vukolic. "Mir-BFT: High-Throughput BFT for Blockchains". In: *CoRR* abs/1906.05552 (2019). arXiv: 1906.05552 (cit. on pp. 2, 86).

[66] D. Steinbeck. "Crypto Snippets: The Stellar Blockchain just Crashed — This is why Nodes Matter". In: *Cryptolawinsider* (2019). URL: https://cryptolawinsider.com/stellar-crash/ (visited on 01/01/2022) (cit. on p. 2).

[67] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006 (cit. on pp. 7–9).

[68] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. "EBAWA: Efficient Byzantine Agreement for Wide-Area Networks". In: *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*. 2010, pp. 10–19 (cit. on pp. 4, 24, 31).

[69] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. "Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary". In: *2009 28th IEEE International Symposium on Reliable Distributed Systems*. 2009, pp. 135–144 (cit. on p. 24).

[70] M. Vukolić. "The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication". In: *Open Problems in Network Security*. Ed. by J. Camenisch and D. Kesdoğan. Cham: Springer International Publishing, 2016, pp. 112–125 (cit. on pp. 18–20).

[71] G. Wood. "Ethereum: A Secure Decentralised Generalised Transaction Ledger". In: *Ethereum project yellow paper* 151 (2014), pp. 1–32. URL: https://files.gitter.im/ethereum/yellowpaper/VIyt/Paper.pdf (visited on 01/01/2022) (cit. on p. 1).

[72] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang. "Blockchain Challenges and Opportunities: A Survey". In: *International Journal of Web and Grid Services* 14.4 (2018), pp. 352–375 (cit. on p. 20).