TÉCNICO LISBOA

Dynamic Trees for Byzantine Consensus Protocols

Tomás Maria de Aquino da Silva de Araújo Pereira

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Luís Eduardo Teixeira Rodrigues
Prof. Miguel Ângelo Marques de Matos

Examination Committee

Chairperson: Prof. João António Madeiras Pereira
Members of the Committee: Prof. Miguel Ângelo Marques de Matos
Prof. Alysson Neves Bessani

October 2024

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First of all, I would like to thank my Thesis advisors Professor Luís Rodrigues and Professor Miguel Matos for welcoming me as their student and for all the support given throughout this work. Their discernment, fruitful discussions and guidance were fundamental to make this work possible and provided me with invaluable learning opportunities. For the same reason, I would like to extend this thanks to all my colleagues at the DPSS group at INESC-ID for all the support they offered me throughout this work.

I would like to thank my mother and my siblings for the words of encouragement and the unconditional support at home, both of which allowed me to get this far.

I would like to thank my partner, Maria Campos, for always standing by my side through thick and thin and for being a source of comfort and support, offering me refuge from the challenges of day-to-day life over these past few years. Thank you for all the good times and for making everything a bit more bearable.

I would like to thank all my friends and friend circles who have accompanied me throughout my academic journey. It is thanks to you that I am who I am today and it is thanks to your unwavering support during both the good and challenging times over these past handful of years that I was able to come this far. Even as times have gotten busier, I cherish all the time I get to spend with all of you. A special thanks goes out to Sofia Bonifácio, Eduardo Claudino, Afonso Lopes, Ana Sequeira, Tiago Camarinhas, Mateus Gaspar, Francisca Caruso, Inês Silva and Rui Dong, amongst others, for always being there for me all these years.

To each and every one of you – Thank you.

This work was partially funded by Fundação para a Ciência e Tecnologia (FCT), using national funds as part of the projects INESC-ID UIDB/50021/2020, DACOMICO (financed by the OE with ref. PTDC/CCI-COM/2156/2021), Ainur (financed by the OE with ref. PTDC/CCI-COM/4485/2021) and ScalableCosmosConsensus.

Abstract

The use of dissemination and aggregation trees allows Byzantine Fault Tolerance (BFT) consensus protocols to increase both their efficiency and scalability, which are key requirements in blockchain applications. The dynamic reconfiguration of a dissemination tree can be a complex task. As a result, most protocols that use dissemination and aggregation trees avoid frequent reconfigurations by using a stable leader policy. Unfortunately, the use of a stable leader is undesirable in blockchain applications, due to equity and censorship concerns. In this work, we propose efficient techniques to support leader rotation and dynamic reconfiguration of dissemination and aggregation trees in BFT consensus protocols. We have applied our techniques to Kauri, a state-of-the-art tree-based consensus BFT algorithm. Through the experimental evaluation, conducted on a real implementation of our solution, we analyse the performance of our proposed mechanisms and show that dynamic reconfiguration can be supported without incurring a significant penalty on the throughput of the system.

Keywords

Distributed Systems; Blockchain; Byzantine Fault Tolerance; Consensus

Resumo

O uso de árvores de disseminação e agregação permite aumentar a escalabilidade e desempenho de protocolos de consenso tolerantes a faltas Bizantinas. Infelizmente, a reconfiguração destas árvores pode ser complexa, o que leva a que muitos protocolos baseados em árvores usem um líder estável, o que nem sempre é desejável, por exemplo, por questões de equidade ou censura de transações. Neste trabalho propomos técnicas eficientes para executar a mudança de líder e reconfigurar dinamicamente as árvores utilizadas pelo algoritmo de consenso. Integrámos estas técnicas no Kauri, um protocolo de consenso baseado em árvores. Através de uma avaliação experimental do protótipo resultante, mostramos que a reconfiguração dinâmica pode ser concretizada sem incorrer numa penalização significativa do desempenho.

Palavras Chave

Sistemas Distribuídos; Blockchain; Tolerância a Falhas Bizantinas; Consenso

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

BFT	Byzantine Fault Tolerance
CHL	Consecutive Honest Leader
DBFT	Democratic Byzantine Fault Tolerance
DLS	Dwork, Lynch, and Stockmeyer
FLP	Fischer-Lynch-Paterson
ISS	Insanely Scalable SMR
LSO	Leader-Speaks-Once
PBFT	Practical Byzantine Fault Tolerance
PoC	Proof-of-Capacity
PoW	Proof-of-Work
QC	Quorum Certificate
RTT	Round-Trip Time
SMR	State Machine Replication
TID	Tree ID
WAN	Wide-Area Network

1

Introduction

Contents

1.1 Motivation

Byzantine Fault Tolerance (BFT) consensus protocols allow correct processes to reach agreement even in the presence of a fraction of malicious processes. BFT protocols have been first introduced for synchronous systems [1] but have been subsequently extended to execute in eventually synchronous settings [2–4]. BFT protocols require multiple rounds of message exchange among participants and are, therefore, costly, both in terms of communication and processing. For this reason, early implementations considered a relatively small number of participants (in the order of dozens) [5]. However, with the

emergence of blockchains and the increasing relevance of large-scale systems based on blockchains, the need to design Byzantine consensus protocols that can scale to hundreds of participants has become a relevant topic. [6]

Practical Byzantine Fault Tolerance (PBFT) [2], one of the first BFT consensus protocols designed for eventually synchronous systems, is a leader-based protocol that uses an All-to-All communication pattern, i.e., the algorithm proceeds in rounds where participants send (and receive) messages to (from) every other participant. Due to the use of this communication pattern, PBFT is inherently non-scalable. Several approaches have emerged to circumvent the scalability limitations of PBFT-like protocols. One approach consists of using a star-based communication pattern, such as HotStuff [3], where nodes only communicate directly with the leader: this reduces the message complexity from quadratic to linear, but the leader remains a bottleneck. Another approach is to use hierarchical strategies, such as in Fireplug [7], where consensus is achieved by the hierarchical combination of several sub-consensus instances that are executed among smaller sub-groups. A limitation to approaches of this nature is how they reduce the system's resilience, given that the fraction of Byzantine nodes required to take control of a sub-group is smaller than the fraction of nodes required to prevent consensus in the super-group. Once a sub-group is compromised, the consensus in the super-group is compromised as well. Finally, the use of dissemination and aggregation trees has been proposed to circumvent the scalability limitations of the star-based communication pattern, while retaining the resilience properties of non-hierarchical approaches. In our work, we focus on protocols that use this strategy. [4, 8–10]

Kauri [4] is a BFT consensus protocol that extensively uses dissemination and aggregation trees to achieve scalability. Since the use of trees introduces extra latency in protocol communication, Kauri uses an aggressive pipelining strategy that expands on the pipelining already used in protocols such as HotStuff, allowing the leader to start multiple consensus instances before previous instances have terminated. Kauri has two main limitations: first, it is designed for homogeneous settings and its trees are constructed using randomization, which does not take into account the different computational capacity of nodes or the latency present in the communication links. Secondly, when a reconfiguration is required, a tree is replaced by a completely different tree that typically does not share any inner node with the previous tree. While this reconfiguration strategy permits Kauri to find a robust tree in a timely manner, it is disruptive to the pipelining process and hence to performance. Additionally, in the case that a robust tree is not found in a predetermined amount of steps, Kauri falls back into a star-topology. As evidenced by these cases, the process of reconfiguring a tree is a complex task, meaning that a vast majority of tree-based consensus algorithms rely on a *stable leader* strategy [4, 8, 9], where the same tree is used for various consecutive consensus instances, only changing when the system fails to make progress (such as the cases where a leader is deemed to be faulty). However, in the context of blockchains, there are various advantages derived from frequently changing the leader node (which, in the case of Kauri,

means changing the tree as well), such as preventing non-apparent malicious nodes from censoring transactions from certain clients.

1.2 Goals

In this work, we address the two main limitations of Kauri identified above. First, we aim to enhance Kauri with a mechanism that would allow the system to diverge from its randomized tree generation and instead opt to use a schedule of different trees throughout execution. This means that the schedule can be adapted towards using trees optimised for heterogeneous networks by leveraging information about the network latencies and the CPU capacity of the nodes. Secondly, we aim to enable Kauri to utilize a rotating leader policy by designing a reconfiguration strategy that can reduce the costs of reconfiguration between two different consecutive trees.

1.3 Contributions

Our thesis proposes novel techniques to dynamically reconfigure dissemination/aggregation trees in BFT protocols, namely, we propose a technique that avoids a significant throughput degradation during the reconfiguration of the dissemination/aggregation tree.

1.4 Results

This work has produced the following results:

i) The implementation of an algorithm that enables Kauri to dynamically change the dissemination/aggregation tree used to reach consensus, according to a predefined schedule that stipulates the topology of each tree and for how long (in blocks) each tree should be used.

ii) An evaluation and analysis of our solution in emulated networks, both in homogeneous and heterogeneous settings.

1.5 Research History

This work has been performed in the context of the DPSS group at INESC-ID. Our work leverages previous results in the design of scalable BFT protocols. Namely, we build on Kauri [4], a BFT consensus protocol for blockchains that has been designed and implemented by Ray Neiheiser, a former PhD student of the DPSS group. Kauri uses a stable leader approach and aimed to design techniques to

support the construction of robust trees. To perform the experimental evaluation of our work, we have used Kollaps [11], a tool that has also been designed and implemented by DPSS researchers. This work has benefited from the help of several members of the Kauri and Kollaps projects.

A preliminary version of this work has been published as:

- T. Pereira, L. Rodrigues and M. Matos. Reconfiguração Dinâmica de Protocolos Tolerantes a Faltas Bizantinas Baseados em Árvores. In INForum24, Lisboa, September 2024.

This work has been supported by Fundação para a Ciência e Tecnologia (FCT), using national funds as part of the projects INESC-ID UIDB/50021/2020, DACOMICO (financed by the OE with ref. PTDC/CCI-COM/2156/2021), Ainur (financed by the OE with ref. PTDC/CCI-COM/4485/2021) and ScalableCosmosConsensus.

1.6 Outline of the Document

The rest of the document is structured as follows: In Chapter 2, we provide the background on relevant blockchain BFT consensus concepts. In Chapter 3, we survey relevant BFT consensus protocols with an emphasis on efficiency and performance. In Chapter 4, we describe our solution's design and implementation. In Chapter 5, we define our evaluation metrics, experimental setup, methodology and goals and discuss our experimental results. Finally, in Chapter 6, we conclude this work.

2

Background

Contents

The objective of this chapter is to introduce the required background to understand our work. Section 2.1 introduces the blockchain abstraction, Section 2.2 defines Byzantine Fault Tolerant Consensus and Section 2.3 discusses the operation of consensus protocols in Wide-Area Network (WAN) deployments.

2.1 Blockchain Systems

The term *blockchain* refers to an abstraction that offers a persistent, append-only, totally ordered ledger of records. Records are grouped into *blocks*, that can be added to the head of the ledger. Once a new block is *committed*, it can no longer be deleted or tampered with. Thus, the sequence of committed blocks is immutable, and the ledger can only be updated by adding additional blocks to the ledger itself. This ledger is typically implemented as a linked list of blocks, where each block has a reference to its

predecessor block in the ledger, thus creating a chain of blocks. Although it is possible to materialize this abstraction using a centralized server, blockchain implementations are generally distributed and decentralized. In this case, the ledger is cooperatively maintained by a possibly large and open set of nodes that coordinate to commit new blocks to the chain and to prevent committed blocks from being tampered with. Using the appropriate protocol, it is possible to derive a reliable and trusted decentralized implementation of the blockchain in settings where a fraction of the participants may not be trusted.

When considering which nodes can participate in a decentralized algorithm to maintain the blockchain, it is possible to classify existing systems into two main categories:

A – Permissionless blockchains: In a permissionless blockchain, any node in the system can participate in the coordination protocol used to maintain the chain. Decentralized ledgers were originally popularized with a permissionless setting, as used in Bitcoin [12]. In this setting, a node can join or leave the protocol at any moment, and the number of nodes participating in the protocol is not limited. Additionally, individual nodes cannot be trusted but it is assumed that a majority of nodes are correct and do not behave maliciously. Malicious nodes can aim to overthrow the system, by discarding blocks proposed by correct nodes and by attempting to prevent correct nodes from proposing new blocks. For this purpose, malicious nodes may also adopt multiple identities to appear as multiple nodes, a behaviour that is often referred to as a Sybil attack [13]. To limit the power of malicious nodes, and in particular, to limit their ability to propose new blocks, a participant must commit a substantial number of resources to the protocol using techniques such as Proof-of-Work (PoW) [12] or Proof-of-Capacity (PoC) [14]. Unfortunately, the use of techniques such as PoW limits the throughput of the system and induces substantial energy consumption, making these protocols ecologically unfriendly. Another limitation of techniques such as PoW is that parties may benefit if they collude to solve the algorithm's crypto-puzzle, which creates an incentive to reduce decentralization.

B – Permissioned blockchains: Permissioned blockchains have gained traction due to their potential to create more efficient and secure systems when compared to permissionless designs [15]. In a permissioned blockchain, participants are known and limited, creating a scenario where the system has controlled decentralization but that also performs better, as mechanisms for Sybil attack resistance are no longer required. Additionally, permissioned blockchains allow for the implementation of BFT consensus protocols, which, combined with the fact that participants have to be authorized, creates a much more controlled environment for system execution. BFT consensus alternatives found in permissioned blockchains are also known to be much more energy efficient when compared to protocols found in permissionless settings, due to the fact that in the latter security is often assured through the use of computationally heavy techniques and algorithms.

2.2 Byzantine Fault Tolerant Consensus

As mentioned above, BFT consensus protocols are a popular solution in permissioned blockchain contexts. The goal of BFT consensus is to allow correct nodes to agree on a singular valid output for an instance of consensus, under the assumption that there may be f nodes exhibiting arbitrary behaviour. While arbitrary behaviour may refer to any type of failure where any error may occur, in the Byzantine Fault Tolerant model we assume the worst-case scenario where faulty nodes may collude with malicious intent to overthrow the system [16].

In a BFT system with N nodes, we define $N = 3f + 1$ to tolerate f Byzantine nodes. This is done by leveraging *quorums* of size $Q = 2f + 1$ for coordination, which guarantees that at least one correct node belongs to any pair of quorums, enabling them to relay protocol messages from one quorum to another. Byzantine Fault Tolerant Consensus is characterized by the following properties [16]:

Definition 1 (Termination)**.** *Every correct process eventually decides some value.*

This is a liveness property. It guarantees that, in every instance of consensus, all correct processes will eventually output a value and thus the system will progress.

Definition 2 (Validity)**.** *If a process decides v, then v was proposed by some process.*

The notion of validity is further extended into two variants in the context of Byzantine Fault Tolerance:

Definition 2.1 (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.*

Definition 2.2 (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 $\square$.*

It is of note that *Strong Validity* does not imply *Weak Validity*; While in Def. 2.1 correct processes can only decide on a value if all processes are correct, in Def. 2.2 correct processes can decide on a value proposed by a correct process or the special value '$\square$' in execution scenarios that may have Byzantine participants. Both variants of Validity can be useful in different contexts, and the task of selecting which one is desired for application execution is up to the protocol designers themselves.

Definition 3 (Integrity)**.** *No correct process decides twice.*

Definition 4 (Agreement)**.** *No two correct processes decide differently.*

Integrity and Agreement, alongside a Validity variant, are the properties that uphold safety throughout consensus execution. When combined with State Machine Replication (SMR), this allows consensus

instances to have a deterministic effect on the system state, ensuring that the agreed-upon changes and commands are propagated consistently to all correct participants.

2.3 Wide-Area Network (WAN) Deployment

Permissioned blockchains are frequently deployed within wide-area networks. As the name implies, WANs are a type of network that connects different devices over a large geographic area, sometimes even globally. WANs may be unreliable and create difficult conditions to guarantee system liveness and achieve good performance, due to the common occurrence of network partitions and due to the usually high latency and low bandwidth that systems must face. These traits accentuate the fact that in most real-world scenarios, network links in a WAN are inherently asymmetric and create a heterogeneous topology, which consequently increases the asynchrony between pairs of nodes. When deployed in these conditions, it is standard for systems to take into account mechanisms that can compensate for and mitigate the downsides of the geographic distance between devices. Blockchains are no exception to this, existing several works that tackle the scalability and reliability of consensus in this setting, such as Steward [17], Mencius [18], and the recently proposed notion of Planetary Scale Byzantine Consensus by Voron *et al.* [19].

For this reason, leader-based protocols when deployed on WAN must have a careful selection of the nodes that will be tasked with leader activities. There is a risk of saturating the leader's network links and additionally, leaders who are outliers in terms of latency might become a bottleneck for system performance. This also applies to Kauri's tree structure, albeit at a more general level since the bottleneck on the leader is alleviated by its topology. Still, as Kauri was originally designed with homogeneous networks in mind, its tree topology ends up being an even greater advantage when adapted to heterogeneous environments, as proven in a recent work [20].

This study of Kauri's performance in heterogeneous environments [20] stems from the fact that, while it may seem that heterogeneous networks may be completely decentralized, permissioned blockchains are often deployed in a few "data center networks". What this means is that a majority of participants can be allocated and pinpointed to a different cluster in the network, which is the structural design basis for cluster-based (or group-based) algorithms, which will be discussed in further detail in the next chapter. Multiple systems tackle the issue of geolocating nodes in order to predict latencies to promote faster communications, such as Vivaldi [21] and AWARE [22]. In the case of Kauri, as a tree topology protocol, different sub-trees may be perceived as different clusters from where the root node branches to. This means that, in theory, tree construction can be enhanced to be informed in regards to the network, when combined with systems like these to create heuristics to promote faster local communication for consensus.

3

Related Work

Contents

Given the background provided by Chapter 2, we can now discuss the relevant features frequently utilized in the design and implementation of state-of-the-art permissioned blockchain BFT protocols. We will first enumerate important traits that help distinguish protocols apart and then look into several examples of BFT protocols that leverage these traits to improve the efficiency and reliability of protocol execution, namely, scalability, reduced communication complexity, reduced latency, and increased system throughput. Finally, we will provide an overarching discussion that will help highlight which beneficial features our solution aims to take advantage of.

3.1 Towards Coordinated Agreement in Blockchain

With the formal definition for BFT consensus defined in Section 2.2, it is important to differentiate and detail some of the more common approaches to achieve agreement between nodes in a distributed system. It is of note that, in a blockchain environment, consensus is run to agree on which transactions will be appended to the distributed ledger and in which order they will be committed and executed. This means that at the heart of consensus applied to blockchain, protocol designs need to take into account mechanisms that ensure coordination between correct nodes for consecutive consensus instances. Through proper coordination, distributed consensus systems are able to commit and execute client requests, avoid conflicts and achieve agreement with reduced message and time complexities [23]. We define the following approaches for coordinated agreement:

A – Leader-based [2–4]. In leader-based BFT consensus algorithms, the role of the coordinator, or as it is usually called, the leader, is to help all correct processes converge towards a decision in a relatively fast manner [24]. In the context of blockchain, it is the leader's role to propose which transactions will be appended to the ledger. Consequently, this means that system progress is directly linked with leader performance. Leaders in consensus are usually defined through the use of *view synchrony* algorithms [16], where the transition from one view to the next dictates a leader change. Due to the importance of the leader role, systems must provide mechanisms to define the leader election policy and to efficiently recover from consensus executions where the leader is found to be faulty. Additionally, in the context of blockchains, the existence of a system-wide leader creates concerns regarding censorship and leader fairness, which has led to the creation of more sophisticated algorithms that aim to mitigate these issues [25, 26]. Leader-based algorithms typically fall under two categories: *efficient leader-based*, where the leader's role is correlated with achieving a target performance with high throughput and low latency in fault-free executions; and *robust leader-based*, where the leader's role is providing high fault-tolerance to the system instead of achieving maximum performance [23].

B – Leaderless [27, 28] Leaderless coordination aims to mitigate leader-based system issues such as censorship and leader-related bottlenecks through the use of higher decentralization [23, 24]. Additionally, by providing equity in regard to the task of transaction proposal, these systems often aim at utilizing this parallelization to obtain gains in the system's transaction throughput. The main disadvantage of leaderless coordination ends up being the complexity of the mechanisms required to reach proper agreement and ordering, especially in the context of blockchains, where blocks need to know the previous block's hash. The lack of a primary node acting as a coordinator in the system creates conflicts related to proposal order. Consequently, solutions end up having increased message and time complexities due to the mechanisms which are usually applied to either avoid or correct such ordering

conflicts.

C – Multi-leader [29, 30]. Multi-leader coordination can be defined as a specification of leaderless design, where instead of giving the ability to propose simultaneously to all system participants, there is a defined set of leaders assigned to a portion of system execution. The multi-leader coordination approach aims at bridging both the advantages of leader-based and leaderless protocols: it utilizes the additional concurrency to achieve a better throughput and avoid bottlenecks while simultaneously limiting the number of leaders in the system to reduce the higher communication overheads and conflict rates seen in leaderless designs. However, unlike leaderless systems, multi-leader protocols must take into account election mechanisms and leader selection policies, which come with additional challenges in terms of algorithm complexity [31].

D – Group-based [7,32]. In group-based coordination, the system is divided into different groups that align with either different responsibilities or different node characteristics, such as geographic location or latency. Afterwards, these groups are used to allow correct nodes to reach agreement through the use of known coordination mechanisms. A common design example of group-based coordination is the definition of a *global* group whose nodes are delegated with the management of a dedicated *local* group. Each of the global group's nodes may perform tasks such as the collection and dissemination of their local group's votes. Consensus is reached through the global group's communication channels and the results are later propagated to the lower hierarchy local groups. Although group-based systems can offer better scalability and performance, these qualities can only be achieved under efficient grouping assumptions, which is a highly complex task when taken into account in a real-world context [31].

3.2 Communication in Blockchain Consensus

Another fundamental aspect of the Byzantine consensus paradigm applied to blockchain is how the participants communicate between themselves throughout protocol execution. We can underline two critical components which can influence both protocol scalability and efficiency:

3.2.1 Timing Assumptions

As expected of most real-world scenarios, distributed systems must be defined within the parameters of real-world network systems, and blockchains are no exceptions. BFT consensus systems usually make one of the three following assumptions regarding the time bounds on communication delays [16]:

A – Asynchronous Systems. The basis for asynchronous systems can be summarized as not making any timing assumptions about processes and links. The passage of time is usually based on the transmission and delivery of messages, opposite to what happens in synchronous systems where participants can accurately time their assumptions by relying on physical clocks. It is still possible to capture some notion of the passage of time by encoding cause-effect relations between events using logical clocks [33]. Asynchronous systems must also take into account the Fischer-Lynch-Paterson (FLP) impossibility result [34], which states that there is no deterministic algorithm that reaches consensus in an asynchronous network when in the presence of at least one faulty node. Several mechanisms have been studied to circumvent this impossibility while retaining an asynchronous system, some of the more popular ones being the use of randomization algorithms for the convergence of consensus values and the termination of consensus instances [15].

B – Synchronous Systems. In the synchronous model, there is a known upper bound for both processing delays (*synchronous computation*) and for message transmission delays (*synchronous communication*). Additionally, synchronous systems are able to coordinate themselves based on time, through the use of a synchronous physical clock in each participant. This local clock has an upper bound on the rate at which it deviates from a global system-wide real-time clock. Due to how difficult it is to implement and assure these timing assumptions, synchronous systems are not common case in the study of BFT consensus in permissioned blockchains.

C – Partially Synchronous Systems. Partial synchrony is derived from the fact that while real-world systems may observe some time bounds (most of the time), these timing assumptions are not guaranteed, creating periods of time where the network is asynchronous. The partial synchrony approach is designed around the fact that the network will correctly hold the proper timing assumptions eventually, just that it is not known when. This makes it so execution is dictated in an eventually synchronous approach, where once it reaches a long enough synchronous period of time, the system will be able to hold timing assumptions, allowing for the remainder of the execution to terminate or do something useful towards system progress. This approach was first introduced by Dwork, Lynch, and Stockmeyer (DLS) [35] to circumvent the FLP impossibility, as a protocol that maintains safety during asynchronous periods will be able to guarantee termination once it reaches a period of synchronous behaviour. These systems are usually complemented with an *eventually perfect failure detector*, which can adjust and leverage the timing assumptions in eventually synchronous systems in order to correctly suspect faulty participants after an unknown amount of time.

3.2.2 Communication Patterns

The adoption of certain communication patterns in protocol execution dictates the message complexity of the different phases of consensus. Such patterns balance the number of communication steps in each phase, which is a source of latency during consensus, with the communication complexity of message propagation itself, which is correlated to bandwidth performance. Some of the more relevant patterns we analyse are as follows:

A – All-to-All. This pattern consists of every process sending protocol messages directly to all other processes during consensus rounds. By avoiding intermediary processes in message delivery, adversaries cannot interfere with the communication of two correct parties, offering robustness. However, this communication pattern often leads to a higher probability of network saturation proportional to the number of participants, as the resulting communication complexity ends up being $O(n^2)$.

B – One-to-All-to-One. In a One-to-All-to-One pattern, a designated coordinator is charged with performing the aggregation of consensus round votes and the dissemination of the results. The aggregation and dissemination of data are two different steps that must be executed at each relevant phase of consensus, resulting in a higher overall latency for each phase when compared to All-to-All patterns. This pattern creates a network topology that is often referred to as a ”logical star”, where the coordinator alleviates the network load at the cost of being a single point of failure with a heavier processing workload. This communication pattern's main advantage is how it reduces the communication to $O(n)$ complexity when compared to standard All-to-All patterns.

C – Tree-based. Tree-based patterns are an extension of One-to-All patterns, where to reduce the load of a singular aggregation and dissemination coordinator, the system instead divides itself into multiple layers of aggregation and dissemination. The root of the tree still executes the functions of a leader, but instead of relaying messages to all other processes, it only does so to its children nodes. Afterwards, each child can be the parent of another set of nodes, performing first the dissemination of information to its children and later the aggregation of the gathered responses to send to its own parent. Thus, this structure spanning a tree spreads the load and responsibility of One-to-All communication amongst all participants charged with the role of parent node, at the cost of further increasing the number of communication steps in each consensus phase. The added latency can be accounted and compensated for by a variety of mechanisms, such as pipelining techniques, which we describe in more detail in Section 3.3.

D – Hierarchical Groups. Hierarchical group patterns, just like tree-based patterns, offer a better distribution of load by assigning the responsibility of information dissemination and aggregation to sets of nodes belonging to specific groups. As previously mentioned in group-based coordination, the usual scenario is the definition of a global group (also called super-group), where each node is delegated the task of information propagation and vote gathering for their respective local group (also called sub-group). Nodes belonging to the super-group all belong to a sub-group, but not all nodes in a sub-group belong to the super-group. The system is then able to achieve consensus on the higher super-group level through the execution of different steps at the sub-group level. The efficiency of this pattern is highly dependent on the quality of the grouping procedure applied to the system, commonly referred to as clustering, as system performance depends not only on the latency between the nodes belonging to the global group but also requires overall low latencies within the local group scope.

3.3 Pipelining

To compensate for the throughput losses experienced by systems with several communication steps in their consensus execution, protocols can leverage the added latency to their advantage through a concept known as pipelining. Pipelining is based on the idea of optimistically initializing future consensus instances while the current one still has not terminated. This can be done due to the fact that each instance of consensus can be divided into several phases (or rounds) of communication that have idle time between them, as nodes have to wait for the propagation and processing of messages in order to receive the replies necessary to advance to the next round. To exemplify, in systems such as Chained HotStuff [3], the leader can optimistically initiate consensus for block $(n+1)$ after it receives the proposal from the previous leader for block n, meaning that it simultaneously executes round 1 of communication for block $(n+1)$ as it executes round 2 of communication for block n. Meanwhile, in Kauri [4], pipelining is further extended by defining the notion of *pipeline stretch*, which exploits the piggybacking functionality of network packets introduced in Chained HotStuff to allow multiple new instances of consensus to be started simultaneously in a singular round of communication of a given consensus instance. Kauri presents a higher pipelining potential when compared to Chained HotStuff due to its nature as a topological tree, where the latency between the root and the leaf nodes creates a considerable amount of idle time for optimistic consensus instantiation. Consequently, pipeline stretch is directly linked with factors related to the tree's configuration: it is dependent on the tree's fanout m (i.e., how many children a parent node has) and on the tree's height h (i.e., the number of layers in the tree).

The application of pipelining is dependent on how often a system may face reconfiguration. In protocols based on a stable leader policy, a designated leader will carry out pipelining tasks throughout multiple instances until it is suspected faulty. The execution of a view-change breaks the pipeline, re-

quiring the following leader to resume it once elected. In the case of rotating leader-based protocols such as Chained HotStuff [3], as each instance is assigned to a different leader, pipelining is performed by making it so each leader executes and certifies only one phase of the protocol and then passes the pipelined information to the following leader. This method of ensuring leader fairness in rotating leader pipelined protocols is often referred to as Leader-Speaks-Once (LSO) [36]. This is facilitated by the fact that HotStuff leverages a concept known as Quorum Certificate (QC), which is used as a collection of votes from a quorum concerning a phase of consensus. That said, QCs can be used to aggregate and disseminate information, and, in the case of Chained HotStuff, generic QCs can be utilized to aggregate information related to concurrent rounds of different consensus instances, simplifying the transfer process of all the necessary information to the subsequent leaders in the pipeline. However, the LSO pipeline approach brings forth liveness concerns due to the existence of the Consecutive Honest Leader (CHL) property [36], which can be summarized as the fact that to reach the successful commit of a single proposal, then there needs to be enough consecutive correct leaders in the pipeline to execute the consensus instance from start to finish. In the case of Chained HotStuff, liveness is hindered by the fact it needs 4 consecutive correct leaders to successfully decide on consensus instances, but pipelining is still proven to be beneficial to the system's overall throughput.

3.4 Reconfiguration

Now that we have established how a system may arrange itself for coordination and communication, it is important to establish the method through which the protocol maintains its desired design choices. Specifically, when dealing with mechanisms such as consensus, it is important to keep track of all the participating members and, in case there is one, keep track of which node may be exercising the role of leader (similarly, in the case of multi-leader systems, we need to keep track of all leader nodes for a given instance). To that effect, we define that consensus is run in a system configuration (usually referred to as *view*) that is updated incrementally when the system deems it necessary to. There are three major reasons why a blockchain consensus system may want to reconfigure: i) to remove a faulty leader from its position; ii) to frequently switch the leader position to ensure transaction fairness and load balancing in the blockchain; iii) for general purpose membership and topology management reasons, such as performance optimisations.

3.4.1 Leader-based Reconfiguration

From the reconfiguration scenarios mentioned above, we can outline two previously mentioned commonplace behaviours found in leader-based BFT protocols that dictate how frequently a system may need to reconfigure itself:

A – Stable Leader Policy. In a Stable Leader Policy setting, the actuating leader is only switched when enough nodes in the system suspect that it is faulty. Once a quorum of nodes deems that the leader is showing arbitrary behaviour, they trigger the reconfiguration mechanism through the exchange of messages to guarantee that the next instance of consensus is run with a new leader. An example of this would be in PBFT [2], where nodes broadcast to all other nodes a *VIEW-CHANGE* message if the current leader is not making enough progress within a given time bound. Stable leader-based systems can leverage this reconfiguration policy to drive higher transaction throughputs in the system as, once a correct leader is picked, it means that the system can avoid the need for reconfiguration for long periods of time in most scenarios. However, in the context of blockchains, this type of behaviour is undesirable as maintaining the same leader for long periods of time raises questions about leader fairness and transaction censorship, as a biased leader can select client requests to be excluded from block proposals for indefinite periods of time. Lastly, Byzantine leaders can greatly diminish the throughput gains of stable leader systems by acting correctly within the bounds of the reconfiguration trigger, at an intentionally slower pace [15]. Such cases require that protocols adopt more fine-tuned and inquisitive fault detection mechanisms.

B – Rotating Leader Policy. Systems based on a Rotating Leader approach usually switch the leader role every consecutive consensus instance, regardless of the perceived leader correctness. Contrasting with the stable leader approach, the Rotating Leader Policy favours a proactive reconfiguration strategy over a reactive one. The chosen leader node is picked according to the protocol's discretion, although as a baseline protocols usually follow a looping pattern that traverses all nodes in the system, such as the one seen in HotStuff [3]. By frequently changing leaders, the system avoids the aforementioned censorship and slow correct leader issues of stable leader based configurations, while simultaneously ensuring better load balancing in regards to the additional workload tied to leader activities. Conversely, this approach incurs an additional time overhead due to how the reconfiguration procedure becomes part of the normal case operation. Additionally, Byzantine leaders will periodically be put in charge of system coordination, hindering system progress.

3.4.2 Reconfiguration Complexity

Given the previous context, it is clear that reconfiguration efficiency is crucial not only when the system is subject to faults but also for normal case operations. To that end, it is important to take into consideration factors that may impact the execution of view change procedures, most notably, the communication complexity behind them. Just as a protocol's execution may follow certain communication patterns, a view change procedure may be designed to follow a similar strategy to enable the system to reach agreement on a new configuration.

We can first mention protocols that utilize an All-to-All broadcast method in order to reconfigure, such as PBFT. When a node suspects that the current leader is faulty, it begins broadcasting a VIEW-CHANGE message to all nodes. Once a quorum of nodes has broadcast this message, if the subsequent view's leader is correct it will eventually prepare and broadcast a *NEW-VIEW* message within the timeout bound. This message shares the needed system state and legitimizes the success of the view change so all nodes may enter the new view. In case of failure, this process is repeated incrementally until a view with a correct leader is found. As expected, albeit robust, this type of reconfiguration pattern is highly limited by its quadratic message complexity, which quickly saturates the bandwidth of the network and offers poor performance in large-scale systems.

For systems based on One-to-All communication such as HotStuff, processes that deem a view change necessary send a NEW-VIEW message directly to the next view's leader. Once this node has aggregated a quorum of NEW-VIEW messages, it can effectively broadcast the collected votes and the necessary system state so that the protocol's next consensus instance runs in the new view. In HotStuff's case, all nodes send a NEW-VIEW message between consensus instances to the next leader as dictated by its rotating leader policy. Overall, this approach has a linear communication complexity, proving to be a more scalable alternative in systems that already leverage a coordinator mechanism. This is the subject of study by solutions such as Cogsworth [37], which aim to make the optimal time of leader-based reconfiguration more common through the use of speedup mechanisms.

3.4.3 Reconfiguration Goal

Another factor which should be taken into consideration is the goal of the reconfiguration. In systems that aren't based on All-to-All communication, where the topology is more complex, the reconfiguration mechanism may need to be run several times in order to reach the desired system state. In the case of Kauri, the system aims at deriving a *robust tree*, which can be defined as [4]:

Definition 5 (Robust Tree)**.** *An edge is said to be safe if the corresponding vertices are both correct processes. A tree is robust iff the leader process is correct and, for every pair of correct processes p_i and p_j, the path in the tree connecting these processes is composed exclusively of safe edges.*

In short, a robust tree can be seen as a logical tree where neither the root nor any internal node is faulty. This is a strict assumption that excludes cases where a tree may lead to consensus even if not robust: for example, trees whose faulty internal nodes have all their children faulty as well. Inherently, the amount of possible tree configurations is considerable in size, while only a small portion of said configurations are robust in nature. As a solution, Kauri leverages this definition to design a reconfiguration mechanism that attempts to find a robust tree configuration in under t attempts, falling back to a star topology in case of failure. This is done by framing the reconfiguration problem as follows: by modelling

a tree configuration as a static graph and the sequence of trees as an *evolving graph*, we can guarantee that there will be a robust static graph under the definition of *recurringly robust evolving graph* [4]:

Definition 6 (Recurringly Robust Evolving Graph)**.** *An evolving graph G is said to be recurringly robust iff robust static graphs appear infinitely often in its sequence.*

Afterwards, Kauri introduces the notion of t*-Bounded Conformity* to define an upper bound on the number of consecutive reconfigurations we might face until we find a robust static graph [4]:

Definition 7 (t-Bounded Conformity)**.** *A recurringly robust evolving graph G exhibits t-Bounded Conformity if a robust static graph appears in G at least once every t consecutive static graphs.*

To reconfigure in t-Bounded Conformity, Kauri shuffles all its nodes into t disjoint bins, each of size equal to or greater than the total number of internal nodes in the system, I. Theoretically, if the number of faulty nodes in the system verifies the condition of $f < t$, then we can guarantee that one of the bins is constituted entirely by correct nodes, meaning that we can draw correct root and internal nodes from a singular bin and assign the leaves utilizing the remaining bins to establish a robust tree. However, in the scenario of a balanced tree with fanout m, there will be at most m bins with size equal to or greater than I. This limits Kauri's reconfiguration to $(m-1)$-Bounded Conformity, where $f < (m-1)$ or more explicitly $f \geq m$ faulty nodes may force the system to instead opt for a star-based reconfiguration with a non-faulty leader. In this worst-case scenario, reconfiguration may take up to $m + f + 1$ attempts and the system loses its tree topology benefits. As this process is randomized, it is of note that Kauri does not aim to preserve its pipeline structure between configurations, opening up optimisation concerns regarding this tree construction algorithm, as is displayed in our work.

3.5 Relevant Systems

To effectively support our solution design on previous BFT protocol contributions, we will first analyse the background and motivation of our declared relevant systems and afterwards describe the key design specifications and mechanisms that bring forth both strong contributions towards our goals and potential pitfalls that warrant careful navigation.

3.5.1 PBFT

Being one of the more well-known BFT algorithms, PBFT was the first protocol to enable systems to tolerate Byzantine faults in a non-synchronous environment [2]. More specifically, while PBFT's safety guarantee is upheld in an asynchronous environment, PBFT still requires at least a partially synchronous

network to guarantee liveness [23]. Based on an All-to-All communication pattern alongside a leader-based design, PBFT reaches consensus in three communication steps. While this design may provide little scalability and high network saturation, PBFT is still one of the most robust BFT protocol definitions and is often used as a building block towards the different solution requirements of a variety of blockchain consensus problems. Protocol execution is as follows (Figure 3.1):

Clients first broadcast their request to all processes in the system. The currently assigned leader process, upon receiving a client request, communicates it by broadcasting a *PRE-PREPARE* message to all processes. This proposal message effectively orders the client's request by assigning it a sequence number. Additionally, it contains the view it is being sent in and the digest of the request. PRE-PREPARE messages are kept small in size as they are used as proof that the request was assigned a sequence number in the specified view during the execution of a view-change procedure. All nodes upon receiving the PRE-PREPARE message acknowledge it by replying with a *PREPARE* message bearing similar contents, which is broadcast to all nodes. When any node receives a Byzantine quorum of $2f + 1$ matching PREPARE messages, it will then proceed with the broadcast of a *COMMIT* message to all nodes once again. Now, when a process receives a Byzantine quorum of COMMIT messages, it will know that consensus was reached as there is a sufficient amount of correct replicas that have also reached the same conclusion. Thus, it can be declared that the client request has been collectively *decided* on and ordered with the sequence number it was assigned in the PRE-PREPARE phase by the leader.

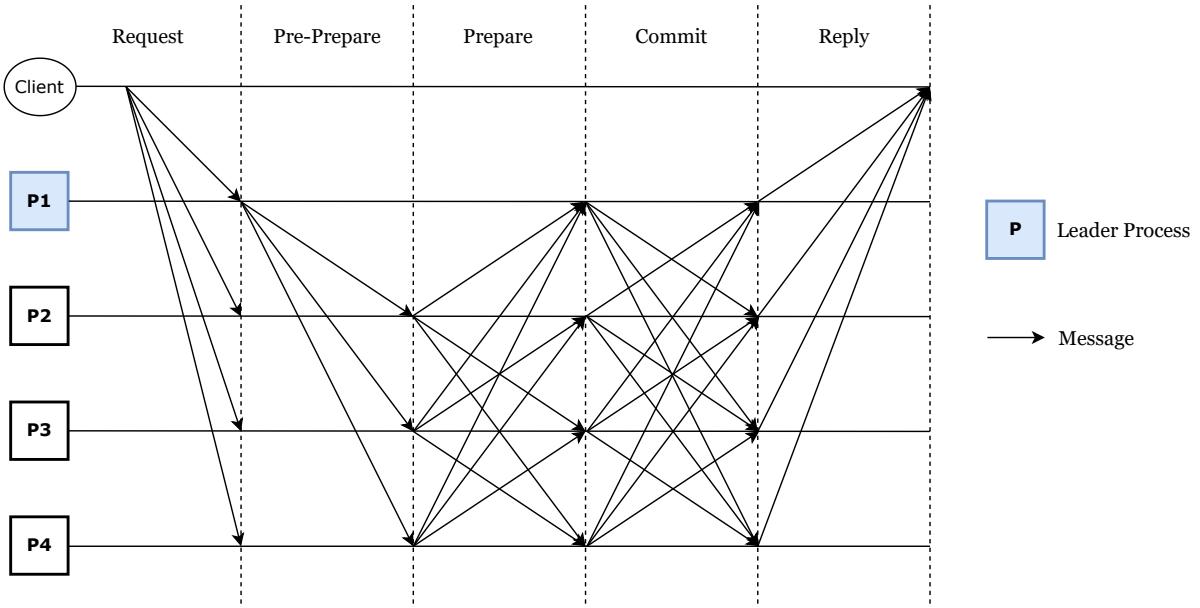

Figure 3.1: PBFT Normal Case Execution

To guarantee liveness, PBFT implements a view-change mechanism to enable progress when the

leader process displays faulty behaviour. In PBFT, views are incrementally numbered, and no two consecutive views have the same leader. The leader-to-view assignment follows a looping pattern, given by $p = v \ mod \ N$, where p is the id of the leader node, v is the number of the view and N is the number of nodes in the system. View changes are typically initiated through the use of a timeout mechanism, where a timer is initialized when processes receive a new request. If the system does not make enough progress within the defined timeout interval, the system first attempts to stay within the same view by doubling the timeout value. If it fails to make progress once more afterwards, then each of the participant's timer is triggered, signalling each process to execute the view change protocol. Just like normal case operations, the view change protocol presents a communication complexity of $O(n^2)$, alongside the added latency invoked by the timeout system, making view changes very costly. Additionally, it has been noted that attackers can collude to arbitrarily delay the system's performance once the timeout reaches a big enough value, making it so only certain client requests are processed as late as possible [15]. To reduce the impact of view changes in system performance, certain algorithms have been proposed to enable broadcast All-to-All systems like PBFT to more commonly reach a theoretical $O(n)$ optimal reconfiguration time, such as the leader-based view change protocol proposed by Naor *et al.* [37].

3.5.2 HotStuff

To promote efficiency and scalability, HotStuff builds on top of PBFT by utilizing the leader to aggregate protocol votes and disseminate results, having a One-to-All communication pattern combined with threshold signatures to guarantee linear communication [3]. HotStuff's goal is to create a protocol based on *optimistic responsiveness* [38] combined with linear communication to reduce the message complexity of not only normal case operations but also view change sequences. For a protocol to be responsive, it means that it can reach consensus in time that is dependant only on the actual message delays instead of being dependent on any known upper bound on message transmission delays [39], the latter being a frequent case in systems based on eventually synchronous models.

To compensate for the heavy workload the leader node faces and to also promote what is often referred to as *chain quality* [40], HotStuff makes it so each consecutive instance of consensus initiates a new view with a new leader from its node rotation, diverging from PBFT's stable leader policy. While this rotating leader approach improves fairness and load balancing, it also creates a time overhead on protocol execution due to the weight and frequency of leader node election [31]. Additionally, on the worst-case of f cascading faulty leaders during leader rotation, the system may be forced to terminate a consensus instance on the magnitude of $O(n^2)$ complexity [23]. Other than the rotating leader behaviour, protocol execution follows PBFT's design, with the additional detail of each phase requiring both the aggregation and dissemination of messages through the leader, as previously described for One-to-All systems in Section 3.2.2. Thus, basic HotStuff execution can be expressed by Figure 3.2.

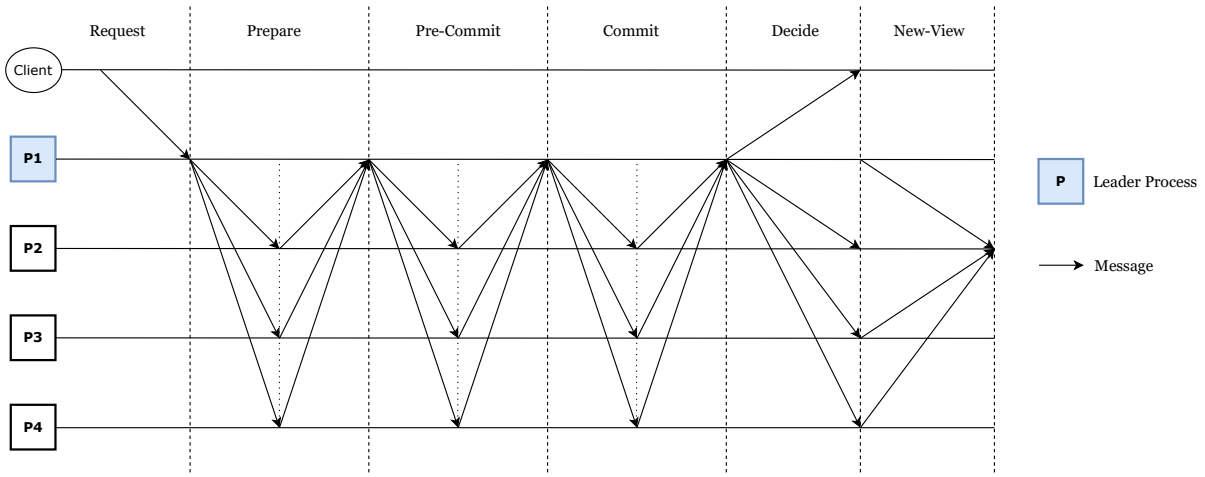

Figure 3.2: HotStuff Normal Case Execution. At the end of the consensus instance, processes send a NEW-VIEW message to the leader of the following consensus instance.

An important concept for HotStuff efficiency is the notion of Chained HotStuff, which utilizes pipelining to mitigate the throughput losses induced by the higher latency of its aggregation and dissemination communication steps. It follows a Leader-Speaks-Once (LSO) paradigm, and, by piggybacking messages related to different consensus instances on the same network packet, Chained HotStuff enables the parallelization of up to four different consensus instances simultaneously. This means that the first round of communication of instance n can be run in parallel with the second round of communication of instance $(n-1)$, the third round of communication of instance $(n-2)$ and the fourth round of communication of instance $(n-3)$.

3.5.3 Kauri

To address the leader bottleneck found in One-to-All systems like HotStuff, Kauri shapes the system's communication pattern into a tree topology with multiple layers of aggregation and dissemination [4]. This way, the workload assigned to the leader node is spread across all nodes that belong to the inner processes of the tree, which execute similar functions to the root except at a sub-tree level. When we say that a node has a fanout of m, it means that it has m children nodes to which it is assigned to disseminate messages to and aggregate the votes of. In the end, this tree structure incentivizes load balancing and improves system scalability, with the trade-off of creating additional communication steps for each layer of depth that the tree has. The protocol's execution can be seen in Figure 3.3. Kauri divides the execution time of a node during a round of consensus into three categories:

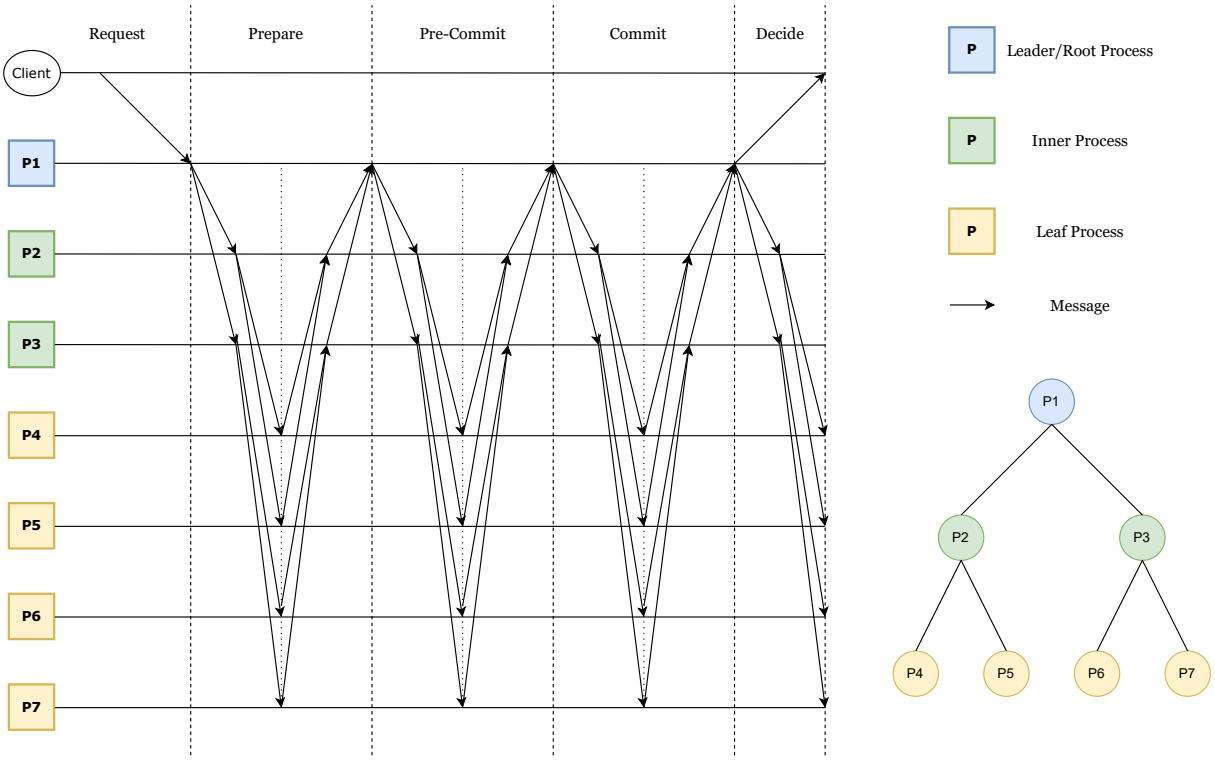

Figure 3.3: Kauri Normal Case Execution with a balanced tree of N=7 nodes.

A – Sending Time. Sending Time is the time a node takes to send (disseminate) a block to all its children. It is dependent on three major factors: node fanout (m), block size (B), and link bandwidth (b). With this, we can say that the time spent on the dissemination process is given by the formula $\frac{mB}{b}$.

B – Processing Time. Processing Time is the time a node takes to validate and aggregate the votes it receives from its children. It is dependent on the fanout m of the aggregating node but it is also heavily dependent on the cryptographic algorithm utilized for signatures. It is given by the formula mP, where P represents the processing time per signature.

C – Remaining Time. Remaining Time is the time that elapses from when the node finishes sending the block to its children until it receives and processes the last reply. To that effect, it depends on the maximum height sub-tree belonging to its children, the Round-Trip Time (RTT) for each layer of said sub-tree, and lastly the processing time of each inner node in the sub-tree. For simplicity, we can consider the case of the root node, which has to take into account the full height of the system tree h. In this case, we have that the remaining time of the root is given by the following formula: $h{\cdot}(RTT{+}Processing\,Time)$.

Inherently, a tree topology-based system like Kauri will have plenty of remaining time in which the

leader is idle. It is through this remaining time that Kauri can effectively calculate its pipeline stretch: by analysing to find out which is bigger, sending or processing time, the system can infer which is the most likely candidate for a bottleneck, the network speed or the CPU, respectively. Afterwards, Kauri calculates the number of consensus instances that can be started during its remaining time through the formula $\frac{Remaining\ Time}{Bottleneck\ Time}$. Note that to reach optimal throughput, Kauri still needs to carefully leverage both the height of the tree and the fanout of the root node, even if it leads to smaller pipeline stretches.

It is important to refer to the fact that the issue of implementing topological awareness in Kauri has been previously discussed [20]. The aforementioned solution proposed an optimistic clustering mechanism based on latencies, with the aim of exploiting the local communication benefits of a balanced tree with C balanced clusters branching from the root. It then leverages this topology-aware clustering approach to implement a rotating leader scenario to provide the necessary chain quality features to Kauri in a WAN deployment.

We however find that this solution has several limitations: the described heuristic is highly optimistic and would gain little to no benefits in an imbalanced cluster scenario. Not only that, the reconfiguration and tree construction approach presented fails to leverage past information in order to provide better support for the future of the system, which is a topic often found in state-of-the-art rotating leader algorithms [25, 36, 41]. It is also important to note that, as the heuristic is solely based on latency, this could lead to repetitive leader node assignments. This is due to how low latency clusters will always have priority during reconfiguration in this scenario, limiting the leader fairness of this solution.

3.5.4 GeoBFT

GeoBFT is a partially synchronous BFT protocol designed to handle geo-scale deployments alongside the ResilientDB permissioned blockchain [32, 42]. The aim is to provide both high scalability and high decentralization benefits to blockchain systems without compromising the throughput of the system. This is achieved through the topologically aware grouping of nodes into local clusters, which lets the protocol distinguish between local and global communication towards a more efficient consensus design. GeoBFT optimistically allows each cluster to make decisions independently and only afterwards relays their decisions through a global channel, thereby globally sharing and ordering all of the locally decided-upon client requests. Effectively, this means that an instance of consensus is separated into three stages: local replication, global sharing and ordering and execution. All of these stages rely on the fact that each cluster has a coordinator named the primary replica. This primary performs the role of leader for local PBFT replication but also performs the role of intermediary with the other primary replicas of the remaining clusters during the global sharing phase.

The notion of independent clusters being able to locally decide on transactions in a decentralized manner harms the robustness of the system as a whole: instead of being able to tolerate the classical

BFT notion of $f = \lfloor \frac{N-1}{3} \rfloor$ faulty nodes in its entirety, the system must verify that condition at a local level for every cluster. As a consequence, for a cluster with N_c total nodes, it can only handle at most $f_c = \lfloor \frac{N_c-1}{3} \rfloor$ local faulty nodes. If this condition is breached, the global sharing phase would compromise blockchain integrity, as other clusters are completely absent from another cluster's local replication process and Byzantine nodes would be able to collude towards proposing their desired block to the global group.

As a partial reconfiguration mechanism, GeoBFT designs a completely decentralized view-change procedure aptly named *remote view-change*. The intuition is that in situations where a Byzantine primary node is correct to every replica except another specific primary replica (where it either does not send messages to or drops all the messages from), it is impossible to know which of the two is Byzantine. To resolve this, we rely on the fact that a cluster C_i may assess the activity of a cluster C_j by communicating not only with C_j's primary node but also with some of its other nodes during the global sharing phase. Likewise, the replies from global sharing are forwarded to a subset of C_i's local cluster nodes. If enough suspicion arises or if communication is deemed unreliable, C_i can reach agreement and request the nodes of C_j to initiate a local view-change to change leader. However, due to the fact that the decision to execute this procedure is external to the cluster, it brings forth a lot of optimisation issues and execution edge cases that make it difficult to be a reliable approach in a real-world implementation, where the multitude of clusters and the complexity of communication may heavily hinder the performance and purpose of the remote view-change, leading to system congestion.

As a last note, it is important to state that GeoBFT does not specify a strategy or metric towards the efficient clustering of replicas at a geographic scale.

3.5.5 Fireplug

Fireplug presents a flexible architecture model for building SMR for geo-replicated databases [7, 43]. Although not initially presented in the context of blockchains, Fireplug proves to be of interest as it tackles both the requisites for scalable BFT consensus and how systems may leverage different kinds of heterogeneity towards their overall robustness to compromises. Fireplug implements geo-scale distribution by defining the existence of data centers and leveraging what it defines as a hierarchical composition of multiple instances of BFT-SMART [44]. This way, each data center communicates locally with a different instance of BFT-SMART and multiple data centers may communicate with each other through a global instance of BFT-SMART. Furthermore, Fireplug abstracts away the interface to interact with each database by implementing both a proxy for multi-versioned data and a replication middleware at a node-based level. This heterogeneity abstraction, coupled with the software diversity provided by its N-Version Programming support, enables Fireplug to leverage diversification at an inter and intra-data center scope. This helps reduce the probability of multiple replicas being compromised simultaneously

through the same vulnerability, whether at a technology or software level.

To coordinate the high amount of possible system configurations, it was later proposed the notion of a replicated *adaptation manager* that maintained the state of each data center through the use of both replicated sensors and actuators, alongside a replica factory. This external manager would enable the system to use global information to deterministically reconfigure single data centers at runtime. However, the kind of centralization that is added by a component such as an adaptation manager would be hard to favour in the context of blockchain, as it would be a critical component for system safety and would need to be assured by a trusted party.

3.5.6 Mir-BFT

Mir-BFT was designed with the idea that the reduction of message complexity is not enough to make a system more scalable and that instead, protocol design should aim to increase throughput to be able to support a higher number of participants [29, 30]. In Mir-BFT, this is achieved by allowing a set of leaders to propose independently and in parallel in an eventually synchronous environment. Additionally, the design choices behind Mir-BFT were adapted into a modular framework named Insanely Scalable SMR (ISS), which can act as a wrapper to enable leader-driven total order broadcast-based protocols to scale and present the same concurrency characteristics as Mir-BFT. However, it is of note that the adaptation of already existing protocols to the ISS framework is not straightforward and often requires compromises that may jeopardize the added concurrency benefits.

To enable the correct ordering of client requests even when there are simultaneous proposals occurring, Mir-BFT multiplexes multiple instances of its broadcast primitive over a partitioned domain of client requests, which can guarantee liveness and safety properties but also implement data duplication prevention mechanisms. The assignment of leaders to the partitioned client hash space is rotated over the course of consecutive *epochs*, meaning that the transition between them can be seen as a view-change in the system. Some of the limitations of Mir-BFT include the high communication costs of the broadcast-based behaviour of the system and the high complexity of the design [31]. An example of the latter is how the batching of client requests depends on concurrent request handling data structures, which is a complex component with a lot of discussion regarding reliable implementation.

3.5.7 Democratic Byzantine Fault Tolerance (DBFT)

DBFT is a leaderless protocol that runs on a partially synchronous network model [27]. In this case, the scalability, load distribution and throughput gains all come from the fact that all nodes play the same role in the execution of consensus, while the offered decentralization avoids bottlenecks. To avoid the FLP impossibility, DBFT utilizes the notion of a *weak coordinator*, whose goal is to help the algorithm

terminate when non-faulty nodes know that their proposals might be decided. This goes against the habitual coordinator behaviour of leader-based systems, where the leader imposes their value on other nodes and forces the system to wait for their decision. It is through this weak coordinator that the system is able to resolve conflicts related to its concurrent proposals. However, for this conflict resolver to work, it requires the existence of partial synchronicity, which is a drawback to the leaderless design of DBFT, as other state-of-the-art leaderless protocols such as HoneyBadgerBFT [45] are able to execute consensus in asynchronous environments.

It was recently argued that this type of leaderless design can drive high throughput and ensure good scalability in WAN settings [19]. This was alleged as DBFT's parallel distinct proposals implement the notion of cumulative All-to-All broadcasts. Through this, the system is able to drive a high payload transfer rate (*goodput*) and exploit larger proposal batch sizes to avoid the waiting time between broadcasts (*hiccup*).

3.5.8 MyTumbler

MyTumbler is a recent timestamp-based leaderless BFT protocol that is able to execute in an asynchronous network model [28]. It aims to create a system that is able to move at the speed of the network delay, where commits are either proposed or aborted, instead of being dependent on the speed of coordinators and the difficulties of tuning the maximum network delay in eventually synchronous systems, where an improperly set timeout bound may lead to excessive consecutive leader changes or too big of a delay to recover from faults. MyTumbler additionally implements the notion of Super Multi-value Agreement, referred to as SuperMA, to leverage an optimal fast path in the randomization process often found in asynchronous protocols. By utilizing a promise mechanism, the non-deterministic common coin [16] component of randomized consensus may be sped up if a quorum of correct processes has submitted the same value for consensus initially. Overall, MyTumbler imposes a trade-off of high communication complexity for the possibility of quick termination, alongside the possibility of transactions having to abort due to conflicts in the asynchronous network.

3.5.9 Other Systems.

We would like to underline the following additional systems, as they might provide supplementary insights to our solution discussion in Chapter 4:

PrestigeBFT [25] is an algorithm which leverages the concept of node reputation (i.e., a metric for the perceived correctness of a participant) to build a system that keeps track of the past faultiness of its nodes. In our context, node reputation is a metric that could be used alongside heterogeneous network performance metrics to better optimise configurations in real-world environments.

ByzCoin [9] is a tree-based algorithm designed for PoW permissionless blockchains, where the participants with higher computational power are able to select which trees to use for consensus. In ByzCoin, the trees are relatively non-robust, due to the use of binary trees, and in the presence of faults, they rapidly degenerate into a *clique* topology, where communication is more costly. Motor [10] improves on ByzCoin by providing an improved tree definition and by designing a rotating leader mechanism to avoid censorship, however, it still suffers from the high latency costs of trees as it does not rely on pipelining techniques as Kauri does. Additionally, Motor's tree construction algorithm is non-optimal as it does not provide a solid definition for tree creation, such as Kauri's robust trees (Definition 5), and its reconfiguration algorithm may incur unnecessary rotations of a tree's sub-trees.

Lastly, we will be referring to Teixeira *et al.*'s attempt at a topologically aware Kauri [20], whose solution was analysed more in-depth in our discussion of Kauri above, as Kauri Enhanced, simplified to Kauri+.

3.6 Discussion

BFT Consensus Algorithm	Coordination Approach	Communication Pattern	Topologically Aware	Leverages Pipelining	Offers $N = 3f + 1$ Resilience
PBFT [2]	Leader-based	All-to-All	✗	✗	✓
HotStuff [3]	Leader-based	One-to-All	✗	✓	✓
Kauri [4]	Leader-based	Tree-based	✗	✓	✓
Kauri+ [20]	Leader-based	Tree-based	**!!**	✓	✓
GeoBFT [32, 42]	Group-based	Hierarchical Groups	✓	✗	✗
Fireplug [7, 43]	Group-based	Hierarchical Groups	✓	✗	✗
Mir-BFT [29, 30]	Multi-leader	All-to-All	✗	✗	✓
DBFT [27]	Leaderless	All-to-All	✗	✗	✓
MyTumbler [28]	Leaderless	All-to-All	✗	✗	✓
Our Approach	Leader-based	Tree-based	✓	✓	✓

Table 3.1: Proposed approach when compared to other existing BFT consensus algorithms. Topological awareness in Kauri+ is highlighted by a **!!** as it is limited.

Table 3.1 summarizes the main characteristics of the state-of-the-art algorithms mentioned above. The features that our comparison aims to highlight are as follows:

A – Coordination Approach. The coordination approach is one of the most integral factors for dictating how a system handles the load distribution between its nodes and if a system may even need to reconfigure at runtime in the first place. The role of coordinator implicitly comes with additional processing costs and responsibilities, meaning that the system may quickly rise in complexity when it is performed by a multitude of nodes. Systems that spread the responsibility of coordination (GeoBFT,

Fireplug, Mir-BFT) or remove the role of dedicated coordinators altogether (DBFT, MyTumbler) often obtain throughput gains from the added concurrency when the communication is non-redundant (distinct proposals). As a drawback, however, the system must implement more complex mechanisms that both deterministically order all concurrent transactions in the system and avoid conflicting simultaneous proposals, which can lead to unsatisfactory design solutions. It is important to note that leaderless systems such as DBFT and MyTumbler do not require the same notion of reconfiguration as the other protocols, since every node has equal capabilities and blockchain membership reconfiguration is out of the scope of their design. Our solution, in order to properly benefit from the load distribution benefits of its tree topology and maintain an easily scalable design, utilizes a leader-based design, now with the additional benefits of a rotating leader policy, which was not the case in the original Kauri.

B – Communication Pattern. A protocol's communication pattern influences the protocol's communication complexity and consequently the system's network saturation and latency as well. We can first note that the All-to-All broadcast behaviour of PBFT, Mir-BFT, DBFT, and MyTumbler can be taxing on the network and processing power of nodes. Other protocols that do not rely on All-to-All communication may have increased node idle times, such as HotStuff and Kauri, as the communication is segmented into several communication steps. In these cases, the use of pipelining techniques compensates for the added latency by optimistically increasing the throughput of the system. Another example of a throughput compensating technique is in GeoBFT, where its highly decentralized hierarchical group pattern allows for its local groups to propose client requests independently and concurrently. Our solution aims to keep the benefits from a tree communication pattern while also keeping the throughput gained from Kauri's extensive pipelining mechanism by optimising reconfiguration so that its disruption on the pipeline is mitigated.

C – Topologically Aware. To be topologically aware is to utilize information regarding how the network or nodes are structured to the system's advantage. The most common example of this is to attempt to use geographic localization to promote communication in faster node links. This usually relies on an external system which enables the clustering or grouping of nodes. Such cases lead to highly decentralized designs, as seen in GeoBFT and Fireplug. Another example of topological conditions that can be leveraged for system efficiency is how computationally strong nodes are. This information allows leader-based, group-based and multi-leader protocols to move coordination responsibilities to nodes which can deliver better performance to the system and mitigate some of the consequences of design bottlenecks. We can state that none of the protocols studied, with the exception of Fireplug, have this capability. While in Kauri+ the topology awareness component ended up being a heuristic applied directly to Kauri's construction algorithm, in our solution we focus on enabling Kauri to reap the benefits of topology awareness through any external system and therefore any desired heuristic. We define a dy-

namic and well-performing reconfiguration algorithm that encapsulates the different configurations Kauri will use throughout execution, enabling Kauri to follow and efficiently rotate through any set of defined trees.

D – Leverages Pipelining. Pipelining is an important factor for increasing consensus performance when in the presence of protocols with ample idle time such as Kauri. Without it, the drawbacks of the additional communication steps of their aggregation and dissemination behaviour would be too costly when compared to the performance of broadcast-based systems such as DBFT. On the other hand, pipelining is a complex mechanism which may need fine-tuning to be able to be fully taken advantage of. An example of this is with the ISS [30] version of HotStuff, where the presence of the modular multi-leader framework alongside pipelining made it so the protocol could not take full advantage of the parallelization, as part of the partitioned proposal space was wasted on dummy batches needed for the extra rounds to avoid breaking the pipeline. Our solution defines a reconfiguration mechanism in a way that it has a reduced impact on pipelining while also allowing protocol designers to choose the best order for consecutive trees, in order to promote an effective transition between two different pipeline structures.

E – Offers $N = 3f + 1$ Resilience. This is a critical aspect of Byzantine consensus, as mentioned in Section 2.2. It is important to highlight this design choice because it is the central weakness of the highly decentralized and highly scalable protocols of GeoBFT and Fireplug. Optimistically, our solution should still maintain the optimal expected resilience while attempting to benefit from the decentralization tactics applied by these two designs. It is important to also highlight that in the case of Kauri, we consider that it has $N = 3f + 1$ resilience due to the fact it exclusively utilizes robust trees for configurations. Internal node faults in Kauri create faulty underlying sub-trees due to the aggregation and dissemination behaviour displayed by its topological tree. However, by utilizing exclusively robust trees, these scenarios are dismissed, therefore allowing Kauri to withstand up to f Byzantine nodes.

4

Dynamic Reconfiguration

Contents

In this chapter, we propose techniques to support dynamic reconfiguration in systems such as Kauri, following the criteria that we have established in Chapter 3 towards the goals defined in Section 1.2. We aim to complement Kauri with a rotating leader policy such that the cost of reconfiguration (more specifically, the impact on the throughput of the pipelining execution) is reduced. As will be verified later in Chapter 5, the performance of our reconfiguration mechanism can be dependent on various factors such as tree height and fanout, the displacement of the nodes between two different consecutive trees (switching two nodes in the leaf layer might have a different effect compared to switching the root node for a leaf node), the frequency of reconfiguration, and so forth. It is also important to emphasize that the overall performance of the system can be influenced by a smart rotation of trees in heterogeneous system contexts, and for that reason, we also provide an analysis of the benefits of our solution in such

environments.

4.1 Model and Assumptions

First and foremost, we recognize that a reconfiguration algorithm must tackle the following challenges:

- a reconfiguration mechanism must have the ability to handle planned reconfigurations (i.e., reconfigurations in a rotating leader schedule) and forced reconfigurations (i.e., reconfigurations that are triggered by faults) in an integrated manner.
- reconfiguration must induce the minimum interference possible in the throughput of the system. In particular, the algorithm must avoid disrupting the pipelining mechanism utilized by systems like Kauri.
- a reconfiguration mechanism must have the ability to handle reconfigurations that can generate situations where participants receive messages that may violate the order of causality.

Given that, we define our model and assumptions as such:

We consider a protocol that implements blockchain services in a permissioned setting, meaning that the group of participants is known amongst themselves. Given this environment, we assume that a tree schedule is provided to all the participants (i.e., a schedule containing the order of the various configurations the system will rotate through), where each tree has its pipeline stretch associated with it (i.e., the number of consensus instances that can be initialized optimistically whilst the current one has not been decided).

We also consider that the system is tolerant to Byzantine faults, where we can support up to $f < \lfloor \frac{N-1}{3} \rfloor$ nodes with arbitrary behaviour from a total of N nodes. The only restriction imposed on the Byzantine nodes is that they do not have the capacity to compromise the cryptographic primitives. The system operates in an eventually synchronous network, where it is possible to guarantee periods of synchronicity between the participants (only it isn't known when), such that it is possible for the system to make progress. During periods of asynchrony, the safety of the system is not compromised.

In this context, we are focusing on deterministic leader-based BFT protocols. The leader communicates with the remaining nodes by utilizing a dissemination and aggregation tree in which the leader is the root node. The protocol executes various instances of consensus, in sequence. We assume that the protocol pipelines these instances optimistically, meaning that it can initialize an instance of consensus before the previous one is terminated. Each instance of consensus requires various rounds of communication to be terminated.

When a consensus instance is initialized by a leader, it uses a given tree to disseminate a block containing client transactions. We denote the tree associated with the said instance as the *initial con-*

figuration. We assume that, in the absence of faults, the initial configuration for all consensus instances is pre-defined. This means that we assume a pre-defined and globally known schedule from which participants obtain the different trees that will be used to execute the first round of communication for each consensus instance.

If, in the absence of faults, all instances use the same configuration, then we consider that the protocol is using a stable leader policy. If not all instances use the same configuration, then we say that the protocol supports *dynamic reconfiguration*. Dynamic reconfiguration can occur whenever a new instance is initialized or periodically. Additionally, the various rounds of communication of a given instance can all use the same configuration for the dissemination and aggregation of values, or, alternatively, use different configurations. In the case of Kauri, in the absence of faults, a consensus instance uses the same tree configuration for all rounds.

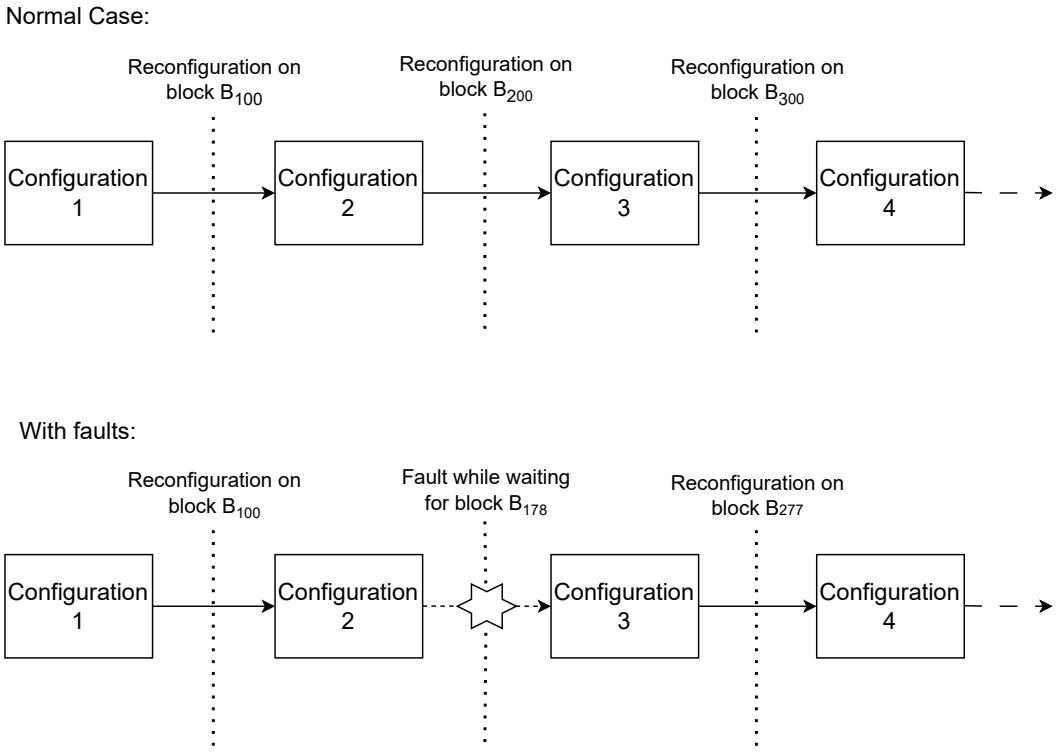

Figure 4.1: Rotations in the case of normal case operations and operations with a fault. We assume all configurations have a duration of 100 blocks.

Lastly, the protocol assumes the same behaviour as Kauri when handling Byzantine faults. This means that dynamic reconfiguration does not affect the original protocol's recovery mechanisms, even during periods of asynchrony or when malicious nodes try to delay the transition between configurations. In the normal case, reconfiguration is triggered in a participant when the last block of a configuration is

delivered. In the case of a Byzantine fault during operations, participants will fail to make progress before delivering the current configuration's last block. In this scenario, participants will move on to the next configuration in the rotation and attempt to make progress under the new tree. The duration of configurations is always fixed, meaning that planned rotations will adapt in the case of recovery so that configurations do the expected amount of blocks. This process takes into account an exponential backoff for its timeout values, in the case of network asynchrony. An example can be seen in Figure 4.1, where on the execution with faults the system must transition to Configuration 3 early due to the fault while waiting for block B_{178}. Future planned reconfigurations are adapted in accordance with this early reconfiguration.

4.2 Schedules

To begin detailing our approach, we first define the input that determines the rotation of configurations utilized in our protocol, specifically tree schedules, which dictate the sequence of trees used during execution. Trees belonging to a schedule are identified from 0 to n, with n being an arbitrary number. Every participant in the protocol can infer from any tree in the schedule the following configuration details: the root of the tree (i.e., the leader), its own parent (if it has one) and its own children (if they exist). Additionally, each tree in the schedule can have a distinct value for its fanout (i.e., the number of children each internal node has) and for its pipeline stretch, making these values dynamic throughout protocol execution. Lastly, each tree has a *target duration*, defined by the number of blocks k during which the tree will remain in effect before the system reconfigures again, assuming the absence of faults. This value can be distinct for every tree in the schedule, although unless stated otherwise we simplistically assume that it is the same.

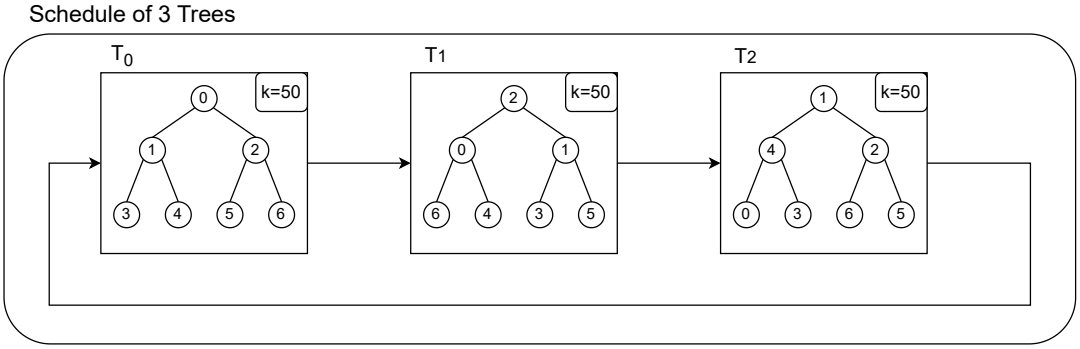

Figure 4.2: Tree Schedule for a system with $N = 7$ nodes. Each tree has a duration of $k = 50$ blocks. Once tree 2 reaches its target, the system reconfigures into tree 0.

When the system reaches the target block of T_n, the system will reconfigure and resume execution

with T_0, making it so that schedules are cyclical. In the current version, in the case of faults, the trees in the schedule are not recomputed. An example of a schedule with 3 trees, each with a duration of $k = 50$ blocks, can be seen in Figure 4.2.

To guarantee equity in the protocol, it is desirable to utilize a schedule of at least N different trees, where each of them has a different participant at the root. Of course, it is possible to consider more complex algorithms for the construction of (sequences of) trees, which take into account the computational resources and the geographic location of nodes, as well as the properties of the network connections like latency, throughout, etc. These types of algorithms are complementary to the contributions of our work, with some preliminary work being done in Kauri+ [20]. Our solution ends up providing the optimised framework to accommodate any schedule of desired trees, whether these are based on informed decisions or randomly allocated.

4.2.1 Advantages of Using Schedules

Given the context of the problem, a rotating leader policy offers additional flexibility in systems that rely on tree topologies, since by allowing the system to choose the structure of the tree, it is selecting which pairs of nodes will establish an edge for communication. Compared to a star topology, where every node is forced to communicate directly with the node that is the leader, in a tree topology we can more easily select the structure that will lead to the usage of communication channels with a better performance in order to drive a higher throughput in the system. However, as the number of possible trees is exponential, exercising a judicious choice of which trees to use is a complex and difficult task, albeit crucial.

Considering that the protocol is to be applied in a permissioned blockchain context, we can assume that nodes can obtain an estimate of the quality of the connections between themselves. Thus, a tree schedule can utilize this information to define trees that can obtain an expected better performance, when compared to randomly chosen trees in a geo-distributed WAN. Additionally, we can further improve the throughput of the system by adapting the pipeline stretch of each tree accordingly and by ordering the trees in the schedule in a way that reduces the impact of the reconfiguration on the execution of the pipelining techniques.

4.2.2 A Simple Schedule

As a base for our rotating leader policy, in order to guarantee that a schedule lets every node perform the role of leader (that is, be the root of a tree), we assume that, unless explicitly stated otherwise, executions will utilize a schedule based on the rotation of the tree's participant array. What this means is that each tree in the schedule is obtained by rotating the nodes in the previous tree. This also means

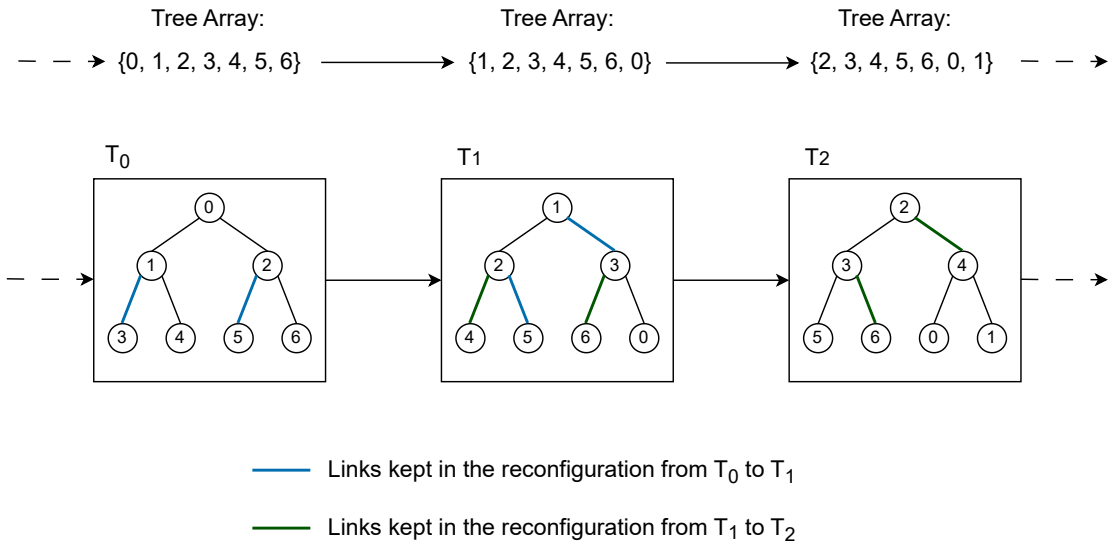

Figure 4.3: A simple rotation-based schedule.

that the schedule will have N trees, with N being the total amount of participants in the system. An example of this type of schedule can be seen in Figure 4.3 for a system with $N = 7$ nodes.

This type of schedule makes it so that each node has an equal opportunity to become and act as the leader of consensus. Each participant is the leader of the tree with an ID that matches its replica ID. In terms of node displacement, this schedule makes it so that consecutive reconfigurations keep part of the links utilized in the previous configuration. Additionally, when reconfiguring, for a tree with h layers, only h nodes will dislocate into a different layer, independently of the fanout. When reconfiguring with this schedule, the previous leader goes to the right-most leaf position, meaning that their workload is alleviated. Meanwhile, the next leader is a direct child of the previous leader, thus it is one of the nodes with the least expected latency to start proposing in the new configuration. All other nodes that are displaced into higher layers of the tree during the reconfiguration process only shift by one layer. This limits the displacement of nodes and therefore the disruption of the throughput of the system throughout the process of reconfiguration, making this schedule a good baseline for our approach.

4.3 Transitioning Between Configurations

One of the biggest challenges in our work is reducing reconfiguration costs. These costs come from various sources, such as the latency caused by the tree's height, the overhead from switching connections and updating the internal state, and the cost of processing blocks that are still in transit through the consensus pipeline during reconfiguration. Additionally, as nodes in higher-up levels receive data earlier

compared to the lower levels of the tree, it means that the time it takes for the system to begin proposing new blocks could be dependent on the level where the next configuration's leader is located. This logic can be applied to the tree as a whole: the more nodes that are displaced from lower levels to higher ones in a reconfiguration, the higher the likelihood that the system will have to wait for them to complete the procedure.

With that said, we know that for any configuration $i \geq 0$ there are k proposed blocks such that B_{ik+j}, where $1 \leq j \leq k$, identifies a block in said configuration. To compensate for the wait mentioned above, we parallelize the transition between two consecutive trees as follows: knowing that block B_{ik+k} is the last block of tree T_i, the leader of the said tree will transition to the new configuration as soon as it finishes proposing B_{ik+k}. Every node that receives block B_{ik+k} will also transition to the new configuration as soon as it verifies if the proposed block is valid and votes for it. Even with nodes transitioning to a new configuration, messages related to the consensus of block B_{ik+k} will be shared by using the same tree that was used for its proposal, meaning that it is decided on the same tree that was used for its proposal. Amongst the nodes that voted for block B_{ik+k} and transitioned to the next tree, if one of them is the leader of the tree T_{i+1}, it starts proposing blocks in the new configuration as soon as it finishes reconfiguration. That being said, the system will finish the pipeline of the previous configuration concurrently with the initialization of the pipeline of the new configuration. This entire process is facilitated by the fact that the protocol messages come with an identifier of the tree that was used for their transmission, allowing nodes to differentiate recipients in communication whenever the system utilizes two trees simultaneously for consensus.

A node that receives block B_{ik+k} but still has not received the blocks in the causal past of B_{ik+k} (i.e., proposals from any consensus instance j, where $ik < j < ik + k$) can only transition to the new configuration once these have been received and processed. Only at that moment can the node process B_{ik+k} and subsequently reconfigure to participate in consensus on tree T_{i+1}. This is because, for a node to reconfigure, it must witness and vote in the proposal that concretizes the target of the configuration in which it is inserted, and for that, it needs to witness and vote in all the blocks in the causal past of said proposal. With that said, although it must witness the last block of the current tree to reconfigure, a node does not need to wait for this block to be decided, meaning that concurrency is increased.

This also applies to proposals from future configurations: in the case that, by reconfiguring, the next configuration's leader proposes a block to a node that has still not reconfigured, this node will wait for the causal past necessary to enter the reconfiguration and only then will it process the pending received proposals (in order). Nodes preemptively partially validate future proposals by confirming if the height of the proposed block is possible (it has to be a height bigger than the target of the current tree) and if the proposing node is indeed the leader of the tree it was proposed in. If these conditions are cleared, the node keeps the proposal pending till reconfiguration. After reconfiguration, a node processes the

pending proposals in order and sends them to their children within the new configuration.

Both these aforementioned edge-cases are exemplified in Figure 4.4: for a system where we reconfigure from T_0 to T_1 on block B_{50}, a node may receive T_0's target block early during the dissemination of the pipeline-stretch (edge-case 1) but it is also possible for a node to receive a proposal from T_1 before witnessing B_{50} (edge-case 2). In this second case, participant $P2$ is the leader of T_1 and starts proposing as soon as it witnesses and votes for B_{50}. As $P6$ is a direct child of $P2$ in T_1 and it still has not received B_{50}, it receives B_{51} before B_{50} and must keep it for later.

Lastly, we provide a formalization of the normal case operation of this algorithm in Algorithm 4.1. Lines $7-15$ provide the leader's propose functionality, assuming that the `Propose` function receives an already validated block of client transactions. Lines $16-28$ provide the remaining replicas' proposal handler logic and its steps for ensuring safety. Note that we must preemptively check pending proposals in line 18 for proposals that fall under edge-case 1 of Figure 4.4, but we can also proactively check them after reconfiguration on line 35 for proposals that fall under edge-case 2. The function `ProcessPendingProposals` follows a similar logic to what is seen in lines $25-34$, however for all the proposals belonging to the *pending_proposals* queue and in-order. This function is formalized later in Algorithm 4.3. Lastly, we can note that this formalization simplifies reconfiguration by decoupling the process from the *Pacemaker* [46], which is an encapsulated fault detector mechanism that ensures liveness in our system implementation-wise. It is through the Pacemaker that reconfiguration is triggered and it is through the Pacemaker that the protocol keeps track of the system's current view, proposer and timeout timers, amongst other things.

Given the intricacies of maintaining a stable pipeline throughput over the course of reconfiguration, it is expected that it is favourable for the system to schedule trees in a way that pairs of consecutive trees differentiate little in their edges. By maintaining similar pipeline structures, the system can reduce wait times for nodes that were previously part of lower layers in the previous configuration. However, this comes with the trade-off of increasing contention for bandwidth on the links that are maintained through reconfiguration.

4.4 Implementation

In this section, we detail the different challenges and mechanisms that our dynamic reconfiguration takes into account to both leverage tree schedules and complement Kauri with a rotating leader policy. The implementation was done in *C++*, over Kauri's public codebase [4, 47], which itself is an adaptation of HotStuff's public codebase [46, 48].

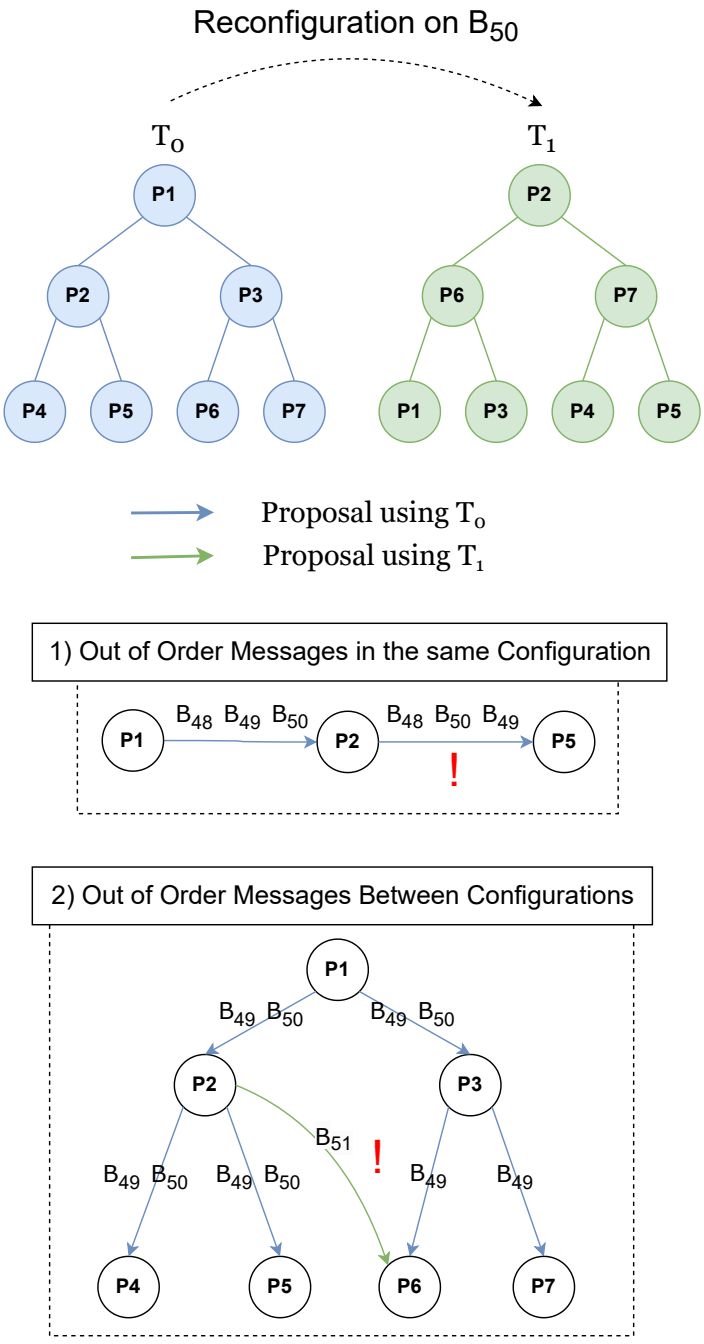

Figure 4.4: Some edge-cases to take into consideration: 1) Nodes that can receive the proposal that concretizes the tree's target out-of-order within the configuration. 2) Nodes that can receive proposals from a future configuration before they reconfigure.

Algorithm 4.1: Commutation to Next Configuration

```
   // Local state for a replica at the start of the protocol
 1 system_trees ← {T_0, T_1, ..., T_n}
 2 current_tree ← T_0
 3 proposer ← GetRoot(current_tree)
 4 child_peers ← GetChildren(current_tree)
 5 lastCheckedBlockHeight ← 0
 6 pending_proposals ← ∅

   // Proposal function for leader
 7 function Propose(block) :
 8     if proposer == GetReplicaId() then
 9         Broadcast(child_peers, PROPOSE(block))
10         Vote(block)
11         lastCheckedBlockHeight ← block.height
12         if block.height == current_tree.target then
13             Reconfigure()
14         end
15     end

   // Proposal handler
16 function ReceivePropose(peer, proposal) :
17     prop_tree ← system_trees[proposal.tid]
18     ProcessPendingProposals()
       // Validate if correct proposer and correct parent
19     if (GetRoot(prop_tree) != proposal.proposer) ∨ (GetParent(prop_tree) != peer) then
20         return
21     end
       // Verify block height
22     if proposal.block.height ≤ lastCheckedBlockHeight then
23         return
24     end
       // Check if we have block's causal history
25     if (prop_tree != current_tree) ∨ (proposal.block.parent.height != lastCheckedBlockHeight)
        then
26         pending_proposals ← pending_proposals ∪{proposal}
27         return
28     end
       // Update internal state
29     Validate(proposal.block)
30     Broadcast(child_peers, proposal)
31     Vote(proposal.block)
32     lastCheckedBlockHeight ← proposal.block.height
33     if lastCheckedBlockHeight == current_tree.target then
34         Reconfigure()
35         ProcessPendingProposals()
36     end
   // Transitions to next configuration in the schedule
37 function Reconfigure() :
38     current_tree ← NextTree(current_tree)
39     proposer ← GetRoot(current_tree)
40     current_tree.target ← lastCheckedBlockHeight + current_tree.duration
```

4.4.1 Challenges

To adjust Kauri to our approach, we must address the following challenges:

- Defining an encapsulated and standardized solution for both schedules and topology trees to facilitate intuitive schedule definition and fast extraction of relevant data.
- Adapting the protocol's messages and message handlers to enable the algorithm to concurrently use connections related to different configurations.
- Modifying Kauri's original Pacemaker in order to distinguish planned reconfigurations from forced reconfigurations. Additionally, we need to define the instant where planned reconfigurations are triggered such that protocol safety is ensured throughout and after the reconfiguration process.
- Enhancing Kauri's out-of-order message handling so that it takes into account messages that may originate from future configurations.

Each of these challenges will be addressed in the following sections, respectively.

4.4.2 Standardized Trees and Schedules

Every replica in the system keeps a data structure in which it will store the currently in-use schedule. This structure is instantiated on system startup, as currently, the protocol utilizes a pre-defined schedule. However, implementation-wise, when reconfiguration is triggered, the system is able to edit the schedule safely throughout reconfiguration, thus allowing for the possibility of the protocol interacting with external components to recompute the schedule at runtime.

Upon startup, the system can either use a default simple rotation schedule with N trees or extract the schedule from a dedicated file. In the file, each line specifies the fanout, pipeline stretch, and participant order for a tree topology in the schedule. Each tree is assigned a Tree ID (TID), either based on the root of the tree (for the default simple rotation schedule) or the order in which the trees appear in the file (if the schedule is extracted from a file).

We define a *Tree* structure (Algorithm 4.2, lines $1-6$), which represents a tree topology objectively for each replica, tracking properties such as the TID, fanout, pipeline stretch, and the tree array (the order of replicas within the tree). This structure is serializable to allow sharing of trees through the replica's communication network library. Additionally, we define a *TreeNetwork* structure (Algorithm 4.2, lines $7-12$), which acts as a wrapper for the Tree structure and calculates relative replica data for the corresponding topology upon instantiation, such as parent and child peers, the number of children in the replica's sub-tree (for aggregation purposes), and more. This abstraction ensures that all relevant data for a given tree is easily accessible and calculated only once.

Algorithm 4.2: Tree Structures for Schedules

```
1  Struct Tree(tid, fanout, pipeline_stretch, duration, tree_array) contains
2      tid
3      fanout
4      pipeline_stretch
5      duration // Used to calculate tree_switch_target
6      tree_array
7  Struct TreeNetwork(ReplicaID, tree) contains
8      tree
9      parent_peer
10     child_peers
11     number_of_children
12     tree_switch_target // Block height for planned reconfiguration.  Calculated every
           time a tree network comes into effect
```

4.4.3 Enabling Concurrent Configurations

To enable Kauri to concurrently handle protocol messages from multiple configurations at once, we need to adapt two key components:

- First, protocol messages must now include the TID of the sender's current configuration (for proposals) or the TID of the message it is replying to. Recipients can use this to quickly identify causality and verify whether the sender is the expected source of the message. For example, in the algorithm, proposals are only relayed by parents to their children. Therefore, if a replica receives a proposal, it can validate the message by checking both: i) whether the proposal originates from the proposer of the tree with the matching proposal TID, and ii) whether the sender is the parent of the replica in the tree with the corresponding message TID (Algorithm 4.1 line 19).

- All message handlers now extract the context of the configuration used to communicate the corresponding message. This is to ensure proper validation and routing of protocol message replies. The process is facilitated by the *TreeNetwork* abstraction, defined in Algorithm 4.2, and can be exemplified in Algorithm 4.1 line 17, where the message's context can be extracted from the *system_trees* map by utilizing the TID contained in the proposal message.

4.4.4 Triggering Reconfigurations

As previously mentioned, both our prototype and Kauri utilize what is known as a Pacemaker to handle liveness and fault detection in the protocol. Implementation-wise, it is the Pacemaker that keeps track of the current configuration and who the current proposer is, and it is also through the Pacemaker that we trigger a reconfiguration in the system.

In normal case operations, planned configurations can be executed by comparing either i) if the proposed block's height reaches the target of the current configuration, in the case of a proposer (Algorithm 4.1, lines $12 - 13$); or ii) if the received proposal block's height reaches the target of the current configuration, in the case of a non-proposer replica (Algorithm 4.1, lines $33 - 34$). This is done after the block is validated and voted for in both cases, maintaining safety, but before the block has been decided, increasing concurrency.

Faults are detected within the Pacemaker component if the system fails to make progress for a determinate amount of time. In these cases, replicas locally advance to the next configuration of the schedule and increase the timeout period. The target of the new configuration can be determined utilizing the tree's known duration and the last block height the Pacemaker recorded. Originally, Kauri's fault handling mechanism considered that once a fault was deemed in a tree, the faulty replica would cease to participate in consensus indefinitely. For our approach, we adapted the Pacemaker to maintain the necessary state for all replicas to continue participating in consensus whether the reconfiguration is forced or planned. The task of removing replicas from the schedule is now encompassed by external schedule computation, meaning that previously faulty nodes stay in the rotation of trees until the schedule is explicitly altered. This task is out of the scope of this work, being further described in our future work section.

Algorithm 4.3: Process Pending Proposals

```
   // We assume that pending_proposals is a queue sorted by the proposals' block
      height
1  function ProposePendingProposals() :
2      for prop in pending_proposals do
3          prop_tree ← system_trees[prop.tid]
4          prop_peers ← getChildren(prop_tree)
           // Check if we have block's causal history
5          if (prop_tree != current_tree) ∨ (prop.block.parent.height != lastCheckedBlockHeight)
            then
6              break
7          end
           // Update internal state
8          Validate(proposal.block)
9          Broadcast(prop_peers, proposal)
10         Vote(proposal.block)
11         lastCheckedBlockHeight ← proposal.block.height
12         if lastCheckedBlockHeight == current_tree.target then
13             Reconfigure()
14         end
15         pending_proposals ← pending_proposals \ {prop}
16     end
```

4.4.5 Out-of-Order Message Handling

While Kauri's original implementation includes detection and handling mechanisms for pipelined blocks that might arrive out-of-order, it must be adapted to take into account that i) an out-of-order block might be the one to reach the target of the current configuration (edge-case 1 in Figure 4.4) and ii) that blocks can be out-of-order due to being from a future configuration (edge-case 2 in Figure 4.4).

By defining a queue for the pending out-of-order blocks at the proposal handling step, we can process them in order using Algorithm 4.3. This function can be safely executed at any time in the protocol, although the key moments where it needs to be called are right before we process a new proposal (Algorithm 4.1 line 18) and right after a reconfiguration (Algorithm 4.1 line 34). In a worst-case scenario, we may have an entire pipeline stretch of proposals from the next configuration pending, which can all be processed in order in a single execution of Algorithm 4.3.

4.4.6 Execution Example

To visualize our prototype's optimized reconfiguration, we provide the following example. In Figure 4.5, we visualize Kauri's pipeline in motion when deployed as a tree of 3 layers ($N = 7, m = 2$). In this scenario, the execution is a stable leader approach and the pipeline is set to have a pipeline stretch of 4, meaning that for each pipeline batch, a block is disseminated alongside 4 optimistically disseminated blocks. This diagram simplifies communication by merging the interactions of nodes within the same layer into one. Realistically, nodes in the same layer diverge in terms of message arrival and relaying times, further extending the time it takes to decide a pipeline batch.

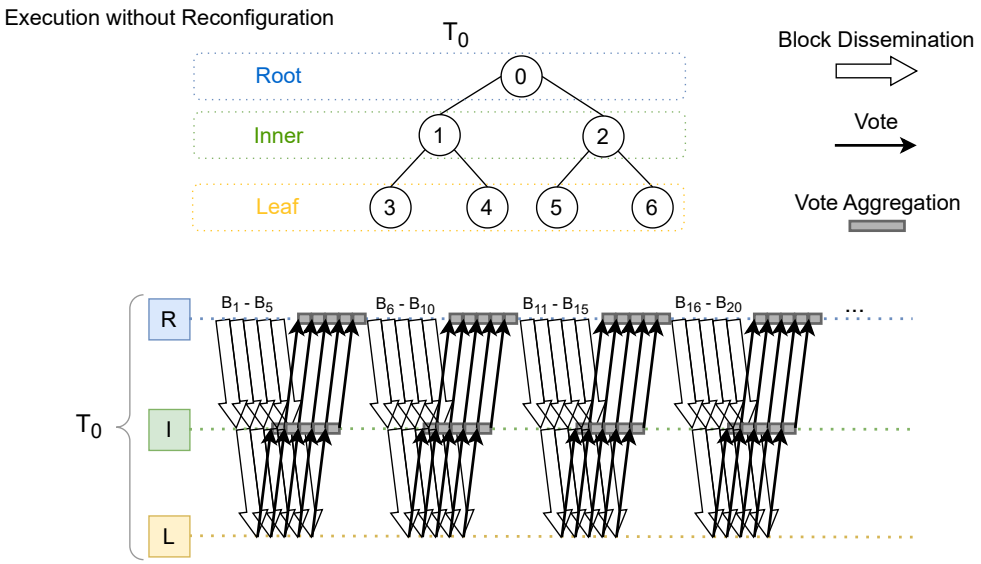

Figure 4.5: An example of Kauri's stable leader pipelined execution.

To exemplify a rotation with our optimised approach, we provide Figure 4.6. We instantiate this execution with our simple rotation schedule (Figure 4.3), and additionally, reconfiguration happens every 50 blocks, meaning that T_0 is reconfigured on block B_{50}. In this case, once nodes receive the disseminated block B_{50} from the root, they can start participating in tree T_1's consensus while simultaneously deciding the previous configuration's remaining blocks. This way, the pipeline is finished in the previous reconfiguration while the pipeline in the new configuration is filled, adding concurrency to our reconfiguration process. We highlight the reconfiguration of the node that is T_1's leader with the purple arrow. The gained concurrency is increased the earlier the next configuration's leader receives the last block of the current configuration. It is important to keep in mind that in larger systems, nodes within the same layer experience greater divergence in message communication times (which is not represented in these diagrams). This accentuates the need for a more thoughtful approach to the placement of the nodes in consecutive tree rotations to better exploit concurrent configurations.

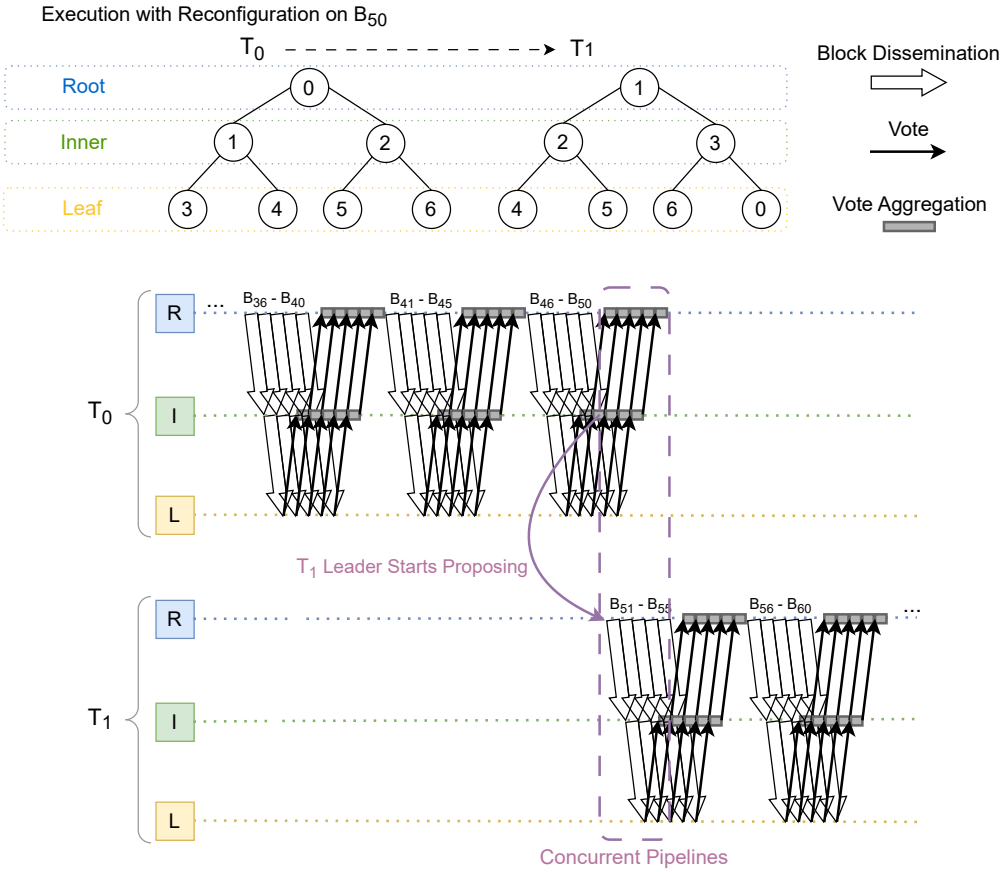

Figure 4.6: An example of dynamic reconfiguration.

4.5 Discussion

While our solution's base rotation schedule may seem promising in terms of equity and efficiency at first, as it gives every node the opportunity to be the leader of a tree in the schedule and keeps the displacement of nodes between configurations reduced, it is important to keep in mind that this kind of schedule may not be desirable for real-world applications. There may be benefits to having schedules that:

1. Have more than N trees, so that certain high-performance trees are included or repeated throughout execution;
2. Have less than N trees, to skip certain configurations with a high risk of either being non-robust or low performance;
3. Have more than N trees, where some consecutive configurations do not change the leader node but instead reconfigure partial sub-trees of the previous configuration;
4. Take into account the equity of consecutive leaders regarding their geographical position;
5. Take into account the geographical location of nodes and reduce the impact of the displacement between two consecutive trees.

With that said, we can state that there are a variety of different heuristics that schedules can follow, and our solution provides the footing that enables Kauri to make use of these to achieve a good performance in WAN networks with leader rotation in mind.

We can say that Kauri's original tree construction algorithm provides what we can call a *randomized schedule*, assuming that we encapsulate the algorithm's generated trees into a limited and cyclical schedule. The rotating leader base proposed for our approach, exemplified in Figure 4.3, is what we call a *rotation schedule*. Lastly, any schedule that orders the nodes in accordance with an external metric or heuristic is what we call an *informed schedule*.

5

Evaluation

Contents

5.1 Evaluation Goals

In this chapter, we present various cases of interest that enable us to analyse the impact of our dynamic reconfiguration in Kauri. To do this, we measure the throughput of the system during the reconfiguration process and compare it to the usual throughput of a stable leader execution with the same parameters (albeit, of course, with no planned reconfiguration in its schedule). This way, we can establish Kauri's original performance as the baseline and analyse whether or not our reconfiguration mechanism exhibits detrimental costs that would justify picking a stable leader over our rotating leader solution.

5.2 Experimental Setup

This experimental evaluation was conducted through the use of Kollaps [11], which permits us to emulate the different characteristics of a distributed network, such as the latency and the bandwidth of the links between the nodes. Every experiment was conducted with two physical machines connected in a way that communication costs between them are insignificant when compared to the emulated topology. The workload we provided puts computational resources close to saturation, meaning that processing time is the bottleneck of the system.

Every network utilized defines its network properties at the link level. Each participant has a link to every other participant. Implementation-wise, network communication leverages concurrency to improve message dissemination, meaning that multiple links can be used throughout the process to reduce the sending time of blocks for consensus. It is possible to saturate node communication links by defining a topology where all of a participant's traffic is routed locally through a switch, making it so that a node's link to its switch is a point of contention and the bottleneck for communication. Since resources are already near saturation at the computational level, we choose networks where each participant has a dedicated link to every other participant, avoiding the interference and contention that would occur on both resources when using a bottlenecked switch-based topology.

We plot our figures through the use of a full execution log of every experiment. Every figure uses the following notation: the throughput of the system is measured using the average number of blocks decided per second over a given period of time. When a reconfiguration occurs, the moment where it is launched is highlighted with a diamond on the figure. To simplify our experimental evaluation, we assume that all the trees in the schedule have an equal target duration of k blocks. Since all reconfigurations will occur every k blocks, systems with higher throughputs will be reconfiguring more frequently.

5.3 Homogeneous Networks

The goal of running experiments in a homogeneous network environment is to limit the number of factors that can influence the throughput of the system throughout the process of reconfiguration. This way, we can more easily isolate the variations in the throughput that originate from the factors that we want to control in our testing. In particular, we study the impact of the following three factors in the performance of our dynamic reconfiguration: i) the height of the trees the system uses; ii) the displacement of the nodes between consecutive configurations and iii) the frequency at which the system reconfigures.

For all the experiments run, with the exception of Figure 5.3, we attributed a latency of *50 ms* and a bandwidth of *750 Kbp/s* to all the connections used between the participants. This way, with Kollaps, we emulate a homogeneous network overlay for all the replicas deployed on our physical machines. The selected bandwidth makes it so that the sending time is still a significant factor for the execution of our

algorithm, even if dissemination is parallelized and optimized.

5.3.1 Tree Height

In this section, we evaluate how the height of the trees being used can affect our reconfiguration procedure. In all the executions presented, our trees have a target duration of $k = 300$ blocks and we utilize a simple rotation schedule, as described in Section 4.2.1. To accurately represent the expected load of a Kauri execution, the pipeline stretch of each scenario was adjusted taking into account the height of the trees used for consensus. This is because, in a tree with more layers, the Remaining Time in the system is bigger, which means that the pipeline stretch must increase to compensate for the additional latency. We analyse the following scenarios:

(a) Height of $h = 2$: using $N = 31$ nodes structured into a star topology of fanout $m = 30$. This scenario corresponds to HotStuff's [3] topology.

(b) Height of $h = 3$: using $N = 31$ nodes structured into trees of fanout $m = 5$.

(c) Height of $h = 5$: using $N = 31$ nodes structured into trees of fanout $m = 2$.

Note that we always opted to use trees that are perfectly balanced. This is because unbalanced trees have extra sources of variation in the results that we want to diminish in this study.

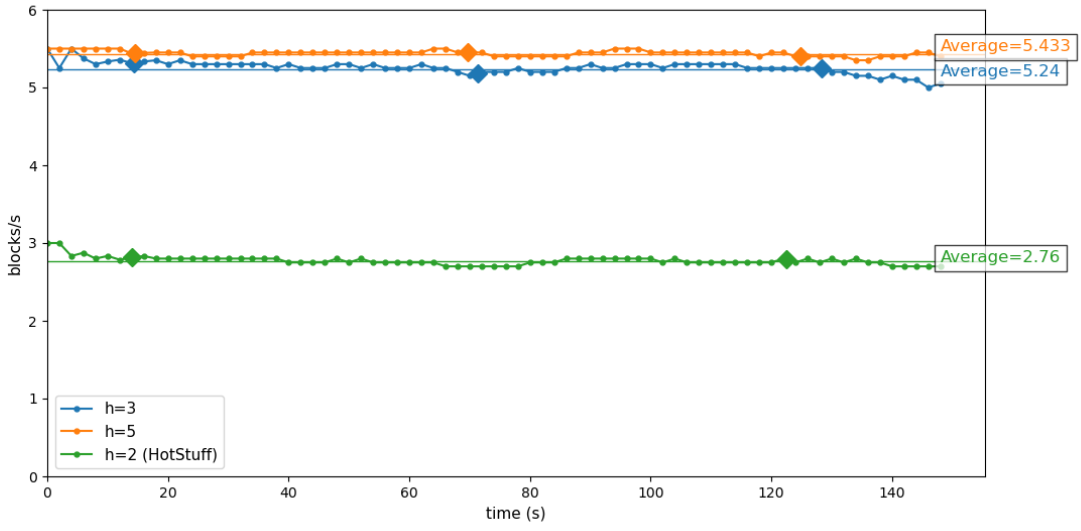

Figure 5.1: Throughput (in blocks per second) over time of the three different scenarios where the tree heights are varied.

In these scenarios, the processing power available for each node is the bottleneck of the execution, further accentuated when the fanout is increased. That said, a star topology execution is the scenario

with the worst performance, even when considering the fact that the implementation can make use of the concurrency offered by our topology to reduce HotStuff's heavy sending time. Meanwhile, both tree topology executions are expected to have a similar throughput, when taking into account that the size of the pipeline for trees with bigger heights is adjusted to compensate for the added latency.

The results of this experiment are represented in Figure 5.1. In the graph, the different scenario executions are aligned so that their first reconfiguration occurs at the same point in time. Additionally, each different scenario has its overall average throughput represented by a horizontal line.

With this experiment, we can deduce the following:

- The overall throughput of the different scenarios follows what was expected regarding the fanout utilized in each of the executions: the scenario that used a HotStuff-like star-based topology for its trees ($h = 2$) has the lowest throughput, as the resources available for a single node are the bottleneck in this experimental setup, being close to saturation. Meanwhile, both executions with higher-depth trees present similar throughputs which are better than HotStuff, as the tree topology balances consensus' workload. Additionally, as the pipelining techniques used were adjusted according to the tree's height, these compensate for the added latency of the trees. The slight difference in performance between trees with $h = 3$ layers and $h = 5$ layers can be justified due to the differences originating from the workload distribution: the trees with fanout $m = 2$ were able to more evenly distribute consensus workload. Although this is a valid strategy for distributing workload in this experiment, it is not applicable to every context as real-world scenarios have to take into account the drawback of more nodes being assigned to internal nodes in the tree, which increases the difficulty of finding a robust tree.

- For this experimental setup and the combination of the selected homogeneous network with the selected tree schedule, the impact of reconfiguration was negligible in all executions. This shows that even when the system is close to saturation and the pipeline is in full use, our reconfiguration mechanism can effectively parallelize the work of two consecutive configurations in a way that the throughput does not drop. In the execution with more layers ($h = 5$), as the pipeline is also increased, the expected additional latency from the layers is compensated during our parallelized reconfiguration process.

5.3.2 Node Displacement

In this section, we evaluate the impact that the displacement of nodes between two consecutive configurations has on the dynamic reconfiguration's performance. For our executions, we use trees with $N = 31$ nodes and a fanout of $m = 5$, meaning that they have a height of $h = 3$. Each tree now utilizes a target duration of $k = 100$ blocks to reconfigure more frequently. To measure the impact of node displacement

in our reconfiguration process, we compare the performance of the reconfiguration mechanism between two different schedule strategies, namely:

(a) **Root-Child Switch Schedule**. This schedule oscillates between two trees, where the root of the next tree is a child of the root of the current tree. It is expected that this schedule has a reduced impact on the reconfiguration process because, by only switching the root with a direct child, the latency of the switch is reduced (as nodes only jump one layer in the tree) and the disruption on the pipeline will be kept to a minimum.

(b) **Internal-Leaf Switch Schedule**. This schedule oscillates between two trees, where the internal nodes of a tree become the leaf nodes of the following tree, and vice-versa (taking into consideration that $\#Leaf\_Nodes > \#Internal\_Nodes$, a sub-group of leaf nodes is picked to swap with the internal nodes). In this schedule, the root of the tree will always be swapped with a leaf node. It is expected that this schedule has a bigger impact on the reconfiguration process, as a high amount of nodes jump various layers in the tree.

Additionally, we include an execution without reconfiguration (i.e., a stable leader execution) to act as the baseline of the expected stable performance. The results are displayed in Figure 5.2.

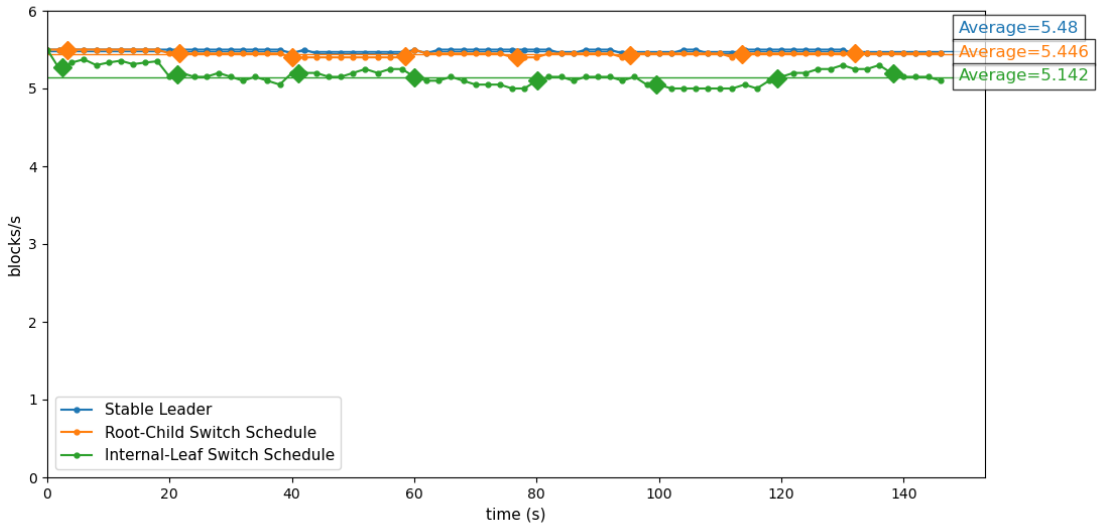

Figure 5.2: Throughput (in blocks/s) over time between two systems with different schedules, when compared to a system that has no reconfiguration.

First and foremost, we can note that the difference in overall average throughput between all executions is minimal, with the biggest one being between the stable leader execution and the Internal-Leaf Switch schedule. The Internal-Leaf Switch schedule execution has roughly 6% less average throughput than the stable leader.

Secondly, as expected, the Internal-Leaf Switch schedule applies more strain to the throughput of the system due to the higher displacement of nodes. This can be seen by the fluctuations of the throughput throughout the execution when compared to Root-Child Switch schedule's more stable throughput. Even then, for such a high displacement of the pipeline structure, the reconfiguration costs are relatively low. This aligns with what was witnessed in the rotating schedule used in Figure 5.1, where the reconfiguration costs were kept at a minimum even when a moderate amount of nodes had to switch their respective parent and children nodes during reconfiguration.

Thus, for schedules that follow Internal-Leaf Switch's behaviour, where most of the nodes change roles upon reconfiguration and the pipeline suffers a bigger disruption, reconfiguration is still feasible without heavy costs. However, we would like to note that the throughput pattern witnessed here is noisier than expected and did not manage to reach the other executions' stable throughput. It is easy to suspect that, if the system reconfigures faster than the throughput stabilizes after the reconfiguration's disruption, then inherently the system will never be able to reach stable performance.

This can be further accentuated by the fact that the system's resources are near saturation. In our experimental setup, nodes are randomly distributed amongst the two physical machines so that each machine has the same amount of nodes. If a tree from the schedule happens to apply more pressure to a single machine by having a majority of its inner nodes located within it, then trees may inherently have a more inconsistent throughput during the recovery process, and thus show a noisier pattern as seen in Internal-Leaf Switch schedule's execution.

For this reason, we provide a follow-up test to this schedule type in Figure 5.3, where the computational load is alleviated and where we give more time between reconfigurations to confirm reconfiguration recovery time.

We once again use an Internal-Leaf Switch schedule, but this time the system only has $N = 21$ nodes laid out in trees of $h = 3$ height where fanout is equal to $m = 4$. For further insight into the high displacement of nodes in this scenario, we vary the latency of the network between the values of *50 ms*, *150 ms* and *250 ms*, with the pipeline stretch adjusted accordingly. There's no change to the bandwidth of the network, and reconfiguration happens every $k = 400$ blocks.

As seen in Figure 5.3, recovery time for reconfiguration is consistent across all networks, with higher latency networks having a stronger throughput impact. Even then, the overall average throughput of all executions is largely unaffected, maintaining similar values. We also note that certain reconfigurations present in this schedule end up having less impact than others, which highlights how important it is for schedules to not only have in mind network properties but also node workload and computational power.

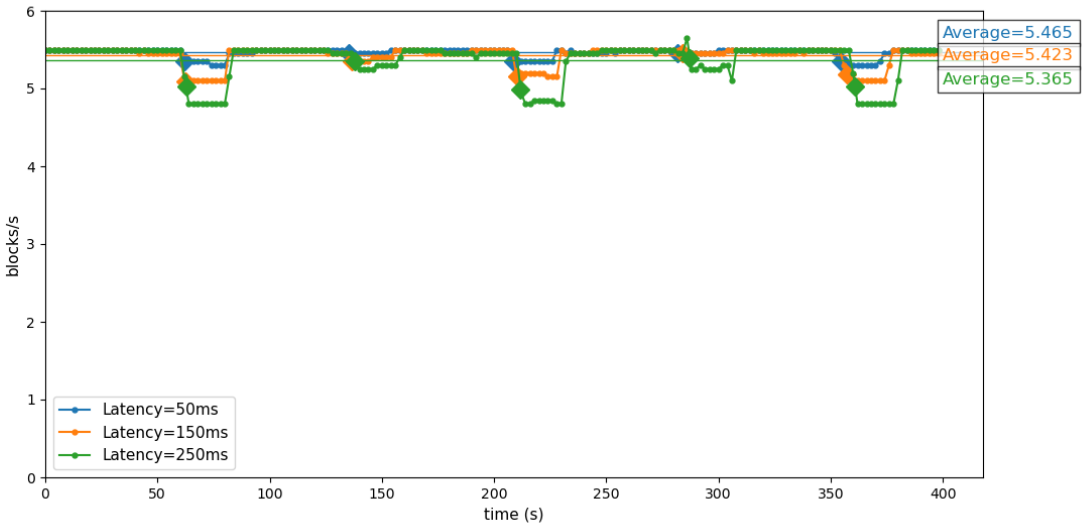

Figure 5.3: Throughput (in blocks/s) over time in Internal-Leaf Switch schedule executions where we vary the latency. This setup utilizes fewer nodes ($N = 21$) to alleviate computational load.

5.3.3 Reconfiguration Frequency

Finally, we study the impact that the frequency of reconfiguration can bring to the performance of the system. We once again use a simple rotation schedule and a system with trees of $N = 31$ nodes with a fanout of $m = 5$. This time, however, we evaluate the performance of our system for four distinct k values, namely:

(a) $k = 1$: A system that rotates leader every block, emulating LSO algorithms like HotStuff [3]

(b) $k = 25$: A system that rotates every 25 blocks.

(c) $k = 50$: A system that rotates every 50 blocks.

(d) $k = \infty$: A system where the configuration is not changed. (Stable Leader)

The results are shown in Figure 5.4. We include the execution that uses HotStuff's topology from Figure 5.1 for comparison purposes.

Once again, for the homogeneous network selected, reconfiguration following the rotation schedule does not seem to provide any noticeable costs for values where k does not interrupt the pipeline. In these cases, there are negligible throughput losses when the reconfiguration rate is higher. Meanwhile, when $k = 1$, Kauri suffers a loss of throughput as, by reconfiguring every block, the system can not make use of its pipelining techniques to compensate for the latency derived from its tree topology. Even then, Kauri still gains throughput in this context over the HotStuff topology seen in Figure 5.1, as it more

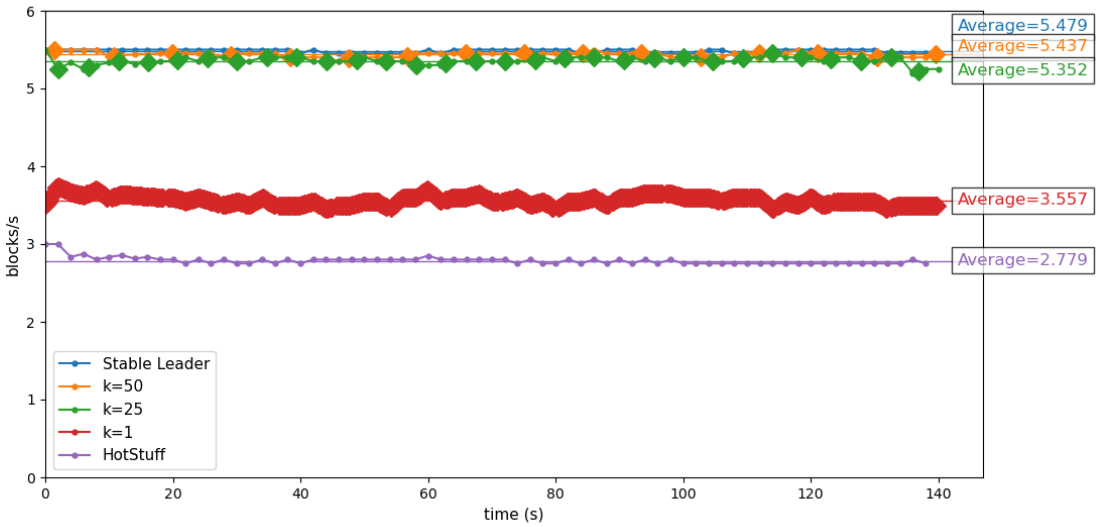

Figure 5.4: Throughput (in blocks/s) over time between four different systems with different reconfiguration frequencies.

evenly distributes the load across various participants. This means that a LSO approach to Kauri can in fact outperform HotStuff when combined with our dynamic reconfiguration.

5.4 Heterogeneous Networks

We can infer the utility of our solution in a WAN by once again utilizing Kollaps, however this time with a careful definition of various network links that could represent a real-world deployment of a permissioned blockchain application. A variety of works take into account that many permissioned blockchains are usually deployed as a data center network, meaning that most participants can be grouped and approximated to different clusters. [32, 49, 50]. Given this, for our heterogeneous experimental evaluation, we define a simplified version of a cluster-based environment for the definition of our network links, represented in Table 5.1:

Clusters	A	B	C
A	50 *ms* 750 *Kbp/s*	150 *ms* 750 *Kbp/s*	250 *ms* 750 *Kbp/s*
B	\	50 *ms* 750 *Kbp/s*	150 *ms* 750 *Kbp/s*
C	\	\	50 *ms* 750 *Kbp/s*

Table 5.1: Table with the network properties between the clusters A, B and C.

We maintain the bandwidth utilized in our homogeneous evaluation meaning that all links in this setup have a bandwidth of 750 *Kbp/s*. This is due to how computational resources are already near saturation, meaning that for our emulated environment, maintaining the previously used bandwidth provides clearer results for interpretation. The most significant aspect of this heterogeneous network definition is not the absolute values chosen for the network properties (which are influenced by and adjusted to our experimental infrastructure), but the relative values between the cluster communication links. The asymmetry brought forth by the latency discrepancies between the clusters is enough to showcase the impact that a heterogeneous network may have on system execution. With this, we can say that network properties for the connections between the three different clusters A, B and C are set so that:

1. Intra-cluster latency is reduced for all clusters, meaning that trees should use same cluster links as much as possible;
2. Latency between A and B is moderate, same with latency between B and C;
3. Latency between A and C is relatively bad, meaning that trees should avoid using links between these two clusters.

We once again use a system with $N = 31$ nodes structured into balanced trees of $m = 5$. Nodes are distributed along the clusters so that $N_A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $N_B = \{10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20\}$ and $N_C = \{21, 22, 23, 24, 25, 26, 27, 28, 29, 30\}$ are the groups of nodes for clusters A, B and C respectively. To evaluate the impact of a smart tree schedule in this heterogeneous network setting, we establish the following three schedule executions:

(a) **Rotation Schedule**: the same rotation schedule utilized in the evaluations of Section 5.3.1 and Section 5.3.3;

(b) **Randomized Schedule**: a schedule of N trees where trees are completely randomized, akin to what Kauri originally used for its bucket-based construction;

(c) **Informed Schedule**: a schedule of N trees that avoid weak links and prioritize intra-cluster communication.

It is important to note that our dynamic reconfiguration allows different trees in the schedule to have different pipeline stretch values. However, here we simplistically assume that all trees have the same ideal pipeline stretch, equivalent to the one used in the homogeneous network evaluation. This is to compare efficiency between the three different schedules more strictly, and because pipeline stretch adjustments would reveal partial knowledge of the system's network, thus belonging only to an informed schedule context. Reconfiguration for these experiments happens every $k = 300$ blocks.

From Figure 5.5, we can infer that:

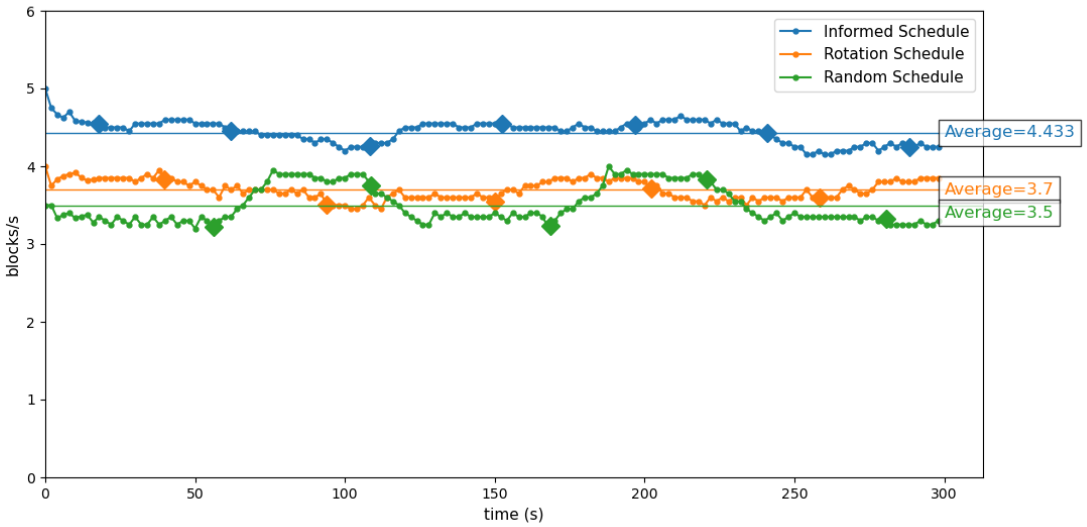

Figure 5.5: Throughput (in blocks/s) over time of three different schedules in a heterogeneous network.

- A random schedule has the worst performance in the heterogeneous network provided. This is because with already 3 different kinds of clusters, the network already has enough variety to increase the probability of inner nodes having high latency links. These network asymmetries can heavily increase the wait experienced to aggregate votes and therefore in consensus.
- A rotation schedule provides a performance that can be compared to a random schedule, however it brings forth more consistency. This is because throughout execution tree performance will often only slightly diverge, as this type of schedule maintains a moderate portion of past links between configurations.
- An informed schedule ends up having better performance than the other two schedules throughout the entire execution. The trees can be set up to avoid weaker links while still shifting around the leadership role between configurations. In theory, informed schedules can be further adjusted in different heterogeneous contexts, such as increasing the pipeline stretch according to the current tree's expected remaining time from its links' latency.

5.5 Discussion

The results of our experiments in a homogeneous network allowed us to infer that dynamic reconfiguration is feasible in consensus protocols based on trees. Figure 5.1 shows that, even with reconfiguration, a tree topology system remains much more efficient than a star topology system, in the case of an environment that bottlenecks the resources of the leader. This means that, when coupled with our solution,

protocols based on aggregation and dissemination trees such as Kauri can offer improvements in system performance in cases where a stable leader is not desirable. Our solution provides high flexibility regarding system requirements and offers good throughput in various scenarios, whether it be frequent rotations, harsher reconfigurations or systems with higher-depth trees.

Meanwhile, the heterogeneous network experiments showcased how, by diverging from Kauri's original randomized tree generation and instead opting to use metrics for building informed robust trees, our dynamic reconfiguration combined with an informed tree schedule can enable even better scalability for WAN deployments.

6

Conclusion

Contents

6.1 Conclusions

With this work, we described a solution that aimed to make Kauri viable with a rotating leader policy. The implementation aims to mitigate the inherent costs of reconfiguring amidst the pipeline execution of this tree-based protocol. Additionally, we set out to define a way to encapsulate the rotation schedule so that system designers can accurately construct the desired tree schedule to be put into use by the protocol. By combining our optimised reconfiguration mechanism with an informed tree schedule, we were able to prove our solution's efficiency in a network that can be approximated to a geo-distributed WAN. Furthermore, by utilizing a homogeneous configuration, we were able to pinpoint how efficient our solution is in a wide variety of scenarios that could affect the dynamic reconfiguration's performance.

Overall, we were able to confirm that Kauri is compatible with a rotating leader policy and that with our solution we can hope to obtain even better scalability in real-world scenarios, whilst obtaining the full

benefits of a rotating leader consensus protocol.

6.2 Future Work

There are several research avenues that we would like to extend our work into:

- **Informed schedule creation at runtime.** Currently, our approach uses a pre-defined schedule throughout the entire execution. When faults are detected within the current schedule tree, Kauri's recovery mechanisms kick in and the system reconfigures towards the next tree. To increase the robustness of the system, it is in our interest to allow an exterior component to take into account faults detected throughout execution to dynamically change the schedule at runtime, so that trees that failed may be recomputed into robust configurations for future rotations. Several protocols have studied how an algorithm can leverage past node correctness to improve and avoid Byzantine leaders, such as Carousel [41], PrestigeBFT [25] and BeeGees [36]. This can be further extended to take into account changes in the network and changes in the performance of certain nodes. Systems like VerLoc [51] have shown promise at geolocating nodes in adversarial settings while systems like AWARE [22,52] and StarReact [53] are able to monitor participant and quorum latency respectively. AWARE aims to provide a prediction model that attempts to minimize consensus latency throughout execution while StarReact is an extension that reacts to network changes that impact quorum certificate aggregation times. These solutions are adequate for a blockchain setting where we aim to build a tree topology that leverages lower latency links for faster communication between participants. Using systems like these as encapsulated components to interact with our dynamic tree schedule creation and recomputation process would enable Kauri to adequately adapt to changes in its environment throughout protocol execution.

- **Tree performance metric.** To complement the monitoring process mentioned above, it is of interest to define heuristics that are able to obtain the expected performance of the different trees in a dynamic schedule. By abstracting the various node and tree performance metrics, we can synthesize a value that would facilitate the selection of trees for an informed schedule. Additionally, this performance value must also take into account the proper ordering inside a schedule in order to minimize the different reconfiguration costs associated with reconfiguration node displacement and waiting times.

Bibliography

[1] L. Lamport, R. Shostak, and M. Pease, *The Byzantine generals problem*. New York, NY, USA: Association for Computing Machinery, 2019, p. 203–226.

[2] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[3] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. Association for Computing Machinery, 2019, pp. 347–356.

[4] 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*. Association for Computing Machinery, 2021, pp. 35–48.

[5] M. Vukolić, "The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication," in *Open Problems in Network Security: IFIP WG 11.4 International Workshop, iNetSec 2015, Zurich, Switzerland, October 29, 2015, Revised Selected Papers*, Springer. Springer International Publishing, 2016, pp. 112–125.

[6] A. I. Sanka and R. C. Cheung, "A systematic review of blockchain scalability: Issues, solutions, analysis and future research," *Journal of Network and Computer Applications*, vol. 195, p. 103232, 2021.

[7] R. Neiheiser, L. Rech, M. Bravo, L. Rodrigues, and M. Correia, "Fireplug: Efficient and Robust Geo-Replication of Graph Databases," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1942–1953, 2020.

[8] W. Li, C. Feng, L. Zhang, H. Xu, B. Cao, and M. A. Imran, "A scalable multi-layer PBFT consensus for blockchain," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1146–1160, 2020.

[9] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing," in *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016, pp. 279–296.

[10] E. Kokoris-Kogias, "Robust and Scalable Consensus for Sharded Distributed Ledgers," *Cryptology ePrint Archive*, 2019.

[11] P. Gouveia, J. a. 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*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

[12] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," *Satoshi Nakamoto*, 2008.

[13] B. N. Levine, C. Shields, and N. B. Margolin, "A Survey of Solutions to the Sybil Attack," *University of Massachusetts Amherst, Amherst, MA*, vol. 7, p. 224, 2006.

[14] S. Park, A. Kwon, G. Fuchsbauer, P. Gaži, J. Alwen, and K. Pietrzak, "SpaceMint: A Cryptocurrency Based on Proofs of Space," in *Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers 22*. Springer International Publishing, 2018, pp. 480–499.

[15] X. Wang, S. Duan, J. Clavin, and H. Zhang, "Bft in blockchains: From protocols to use cases," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–37, 2022.

[16] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Springer Science & Business Media, 2011.

[17] 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," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 1, pp. 80–93, 2008.

[18] C.-S. Barcelona, "Mencius: building efficient replicated state machines for WANs," in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, 2008.

[19] G. Voron and V. Gramoli, "Planetary Scale Byzantine Consensus," in *Proceedings of the 5th Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed Systems*. Association for Computing Machinery, 2023, pp. 1–6.

[20] H. Teixeira, L. Rodrigues, and M. Matos, "Árvores de Disseminação e Agregação Cientes da Topologia para Suportar Consenso Bizantino em Larga Escala," *INForum*, 2023.

[21] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, “Vivaldi: A decentralized network coordinate system,” *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 15–26, 2004.

[22] C. Berger, H. P. Reiser, J. Sousa, and A. Bessani, “AWARE: Adaptive wide-area replication for fast and resilient Byzantine consensus,” *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1605–1620, 2020.

[23] G. Zhang, F. Pan, M. Dang’ana, Y. Mao, S. Motepalli, S. Zhang, and H.-A. Jacobsen, “Reaching Consensus in the Byzantine Empire: A Comprehensive Review of BFT Consensus Algorithms,” *arXiv preprint arXiv:2204.03181*, 2022.

[24] K. Antoniadis, J. Benhaim, A. Desjardins, E. Poroma, V. Gramoli, R. Guerraoui, G. Voron, and I. Zablotchi, “Leaderless consensus,” *Journal of Parallel and Distributed Computing*, vol. 176, pp. 95–113, 2023.

[25] G. Zhang, F. Pan, S. Tijanic, and H.-A. Jacobsen, “PrestigeBFT: Revolutionizing View Changes in BFT Consensus Algorithms with Reputation Mechanisms,” *arXiv preprint arXiv:2307.08154*, 2023.

[26] G. Tsimos, A. Kichidis, A. Sonnino, and L. Kokoris-Kogias, “HammerHead: Leader Reputation for Dynamic Scheduling,” *arXiv preprint arXiv:2309.12713*, 2023.

[27] T. Crain, V. Gramoli, M. Larrea, and M. Raynal, “DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains,” in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE, 2018, pp. 1–8.

[28] S. Liu, W. Xu, C. Shan, X. Yan, T. Xu, B. Wang, L. Fan, F. Deng, Y. Yan, and H. Zhang, “Flexible Advancement in Asynchronous BFT Consensus,” in *Proceedings of the 29th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2023, pp. 264–280.

[29] C. Stathakopoulou, T. David, and M. Vukolic, “Mir-BFT: High-Throughput BFT for Blockchains,” *arXiv preprint arXiv:1906.05552*, p. 92, 2019.

[30] C. Stathakopoulou, M. Pavlovic, and M. Vukolić, “State machine replication scalability made simple,” in *Proceedings of the Seventeenth European Conference on Computer Systems*. Association for Computing Machinery, 2022, pp. 17–33.

[31] W. Zhong, C. Yang, W. Liang, J. Cai, L. Chen, J. Liao, and N. Xiong, “Byzantine Fault-Tolerant Consensus Algorithms: A Survey,” *Electronics*, vol. 12, no. 18, p. 3801, 2023.

[32] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, “ResilientDB: Global Scale Resilient Blockchain Fabric,” *arXiv preprint arXiv:2002.00160*, 2020.

[33] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Commun. ACM*, vol. 21, no. 7, p. 558–565, jul 1978.

[34] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[35] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[36] N. Giridharan, F. Suri-Payer, M. Ding, H. Howard, I. Abraham, and N. Crooks, “BeeGees: stayin’ alive in chained BFT,” in *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*. Association for Computing Machinery, 2023, pp. 233–243.

[37] O. Naor, M. Baudet, D. Malkhi, and A. Spiegelman, “Cogsworth: Byzantine View Synchronization,” *Cryptoeconomic Systems*, vol. 1, no. 2, oct 2021.

[38] R. Pass and E. Shi, “Thunderella: Blockchains with Optimistic Instant Confirmation,” in *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part II 37*. Springer International Publishing, 2018, pp. 3–33.

[39] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer, “Bounds on the time to reach agreement in the presence of timing uncertainty,” *Journal of the ACM (JACM)*, vol. 41, no. 1, pp. 122–152, 1994.

[40] J. Garay, A. Kiayias, and N. Leonardos, “The Bitcoin Backbone Protocol: Analysis and Applications,” in *Annual international conference on the theory and applications of cryptographic techniques*. Springer International Publishing, 2015, pp. 281–310.

[41] S. Cohen, R. Gelashvili, L. K. Kogias, Z. Li, D. Malkhi, A. Sonnino, and A. Spiegelman, “Be Aware of Your Leaders,” in *International Conference on Financial Cryptography and Data Security*. Springer International Publishing, 2022, pp. 279–295.

[42] S. Gupta, S. Rahnama, and M. Sadoghi, “Permissioned Blockchain Through the Looking Glass: Architectural and Implementation Lessons Learned,” in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2020, pp. 754–764.

[43] 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)*. IEEE, 2018, pp. 61–71.

[44] A. Bessani, J. Sousa, and E. E. Alchieri, “State machine replication for the masses with BFT-SMART,” in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 355–362.

[45] 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*. Association for Computing Machinery, 2016, pp. 31–42.

[46] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT Consensus in the Lens of Blockchain," 2019.

[47] R. Neiheiser. (2021) Kauri-public. https://github.com/Raycoms/Kauri-Public. Accessed: 2024-10-08.

[48] M. Yin and D. Malkhi. (2019) libhotstuff. https://github.com/hot-stuff/libhotstuff. Accessed: 2024-10-08.

[49] J. Qi, X. Chen, Y. Jiang, J. Jiang, T. Shen, S. Zhao, S. Wang, G. Zhang, L. Chen, M. H. Au *et al.*, "Bidl: A High-throughput, Low-latency Permissioned Blockchain Framework for Datacenter Networks," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2021, pp. 18–34.

[50] M. J. Amiri, D. Agrawal, and A. El Abbadi, "SharPer: Sharding Permissioned Blockchains Over Network Clusters," in *Proceedings of the 2021 International Conference on Management of Data*. Association for Computing Machinery, 2021, pp. 76–88.

[51] K. Kohls and C. Diaz, "{VerLoc}: Verifiable Localization in Decentralized Systems," in *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022, pp. 2637–2654.

[52] M. Nischwitz, M. Esche, and F. Tschorsch, "Raising the AWAREness of BFT Protocols for Soaring Network Delays," in *2022 IEEE 47th Conference on Local Computer Networks (LCN)*. IEEE, 2022, pp. 387–390.

[53] M. Esche, M. Nischwitz, and F. Tschorsch, "StarReact: Detecting Important Network Changes in BFT Protocols with Star-Based Communication," in *2024 IEEE 49th Conference on Local Computer Networks (LCN)*. IEEE, 2024, pp. 1–7.