Self-Adapting BFT Consensus: Leveraging Heterogeneity in Dissemination/Aggregation Trees.

Helena Carolina Delgado Teixeira

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. Rui Filipe Fernandes Prada
Members of the Committee: Prof. Miguel Ângelo Marques de Matos
Prof. Hervé Miguel Cordeiro Paulino

November 2023

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

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

I would like to thank my parents for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible. I would also like to thank my grandparents, aunts, uncles and cousins for their understanding and support throughout all these years.

I would also like to acknowledge my dissertation supervisors, Prof. Luís Rodrigues and Prof. Miguel Matos, for their insight, support and sharing of knowledge that has made this Thesis possible.

This work was supported by FCT – Fundação para a Ciência e a Tecnologia, through the projects UIDB/50021/2020, DACOMICO (PTDC/CCI-COM/2156/2021) and Ainur (PTDC/CCI-COM/4485/2021), and developed within the project no. 51 “BLOCKCHAIN.PT - Agenda Descentralizar Portugal com Blockchain”, funded by European Funds, namely “Plano de Recuperação e Resiliência - Componente 5: Agendas Mobilizadoras para a Inovação Empresarial”, included in the funding program NextGenerationEU.

Last but not least, to all my friends and colleagues who helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.

To each and every one of you – Thank you.

Abstract

Permissioned blockchains are a class of blockchains where the processes that run consensus are limited and known by all participants. These blockchains can execute variants of *Byzantine consensus* that offer *finality*. An approach to support a large number of participants in this context is to use dissemination and aggregation trees to support the communication required to execute protocol rounds. Previous work using this topology is either topology-agnostic or assumes homogeneous environments. Many Byzantine consensus protocols are leader-based, and in the blockchain scenario, there are compelling reasons to rotate the leader between consecutive consensus instances, such as the distribution of the load of the leader and censorship resistance. This work addresses the challenges of implementing a rotating leader policy in BFT consensus that uses dissemination and aggregation trees, and we propose topology-aware heuristics to create dissemination and aggregation trees in heterogeneous environments. Through simulations, we evaluate the performance of our heuristics in realistic scenarios, showing that they can reduce the average time needed to collect a quorum by 70%.

Keywords

Byzantine Fault Tolerant Consensus; Dissemination and Aggregation Trees; Heterogeneity.

Resumo

Blockchains com permissões são uma classe de blockchains em que os processos que executam o consenso são conhecidos por todos os participantes. Estas cadeias de blocos podem executar variantes de *consenso bizantino* que oferecem *finalidade*. Uma abordagem para suportar um grande número de participantes neste contexto é utilizar árvores de disseminação e agregação para suportar a comunicação necessária para executar rondas de protocolo. O trabalho anterior ou é agnóstico à topologia, ou pressupõe ambientes homogéneos. Muitos protocolos de consenso bizantino são baseados em líderes e, no cenário de blockchain, há razões convincentes para girar o líder entre instâncias de consenso consecutivas, como a distribuição da carga do líder e a resistência à censura. Este trabalho aborda os desafios da implementação de uma política de líder rotativo no consenso BFT que usa árvores de disseminação e agregação com heurísticas cientes da topologia para criar árvores de disseminação e agregação em ambientes heterogéneos. Através de simulações, avaliamos o desempenho das nossas heurísticas em cenários realistas, mostrando que podem chegar a reduzir em 70% o tempo médio necessário para a recolha de um quórum.

Palavras Chave

Consenso Bizantino; Árvores de Disseminação e Agregação; Heterogeneidade;

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

BFT	Byzantine Fault Tolerance
PoW	Proof-of-Work
BFT-SMR	Byzantine Fault Tolerant State Machine Replication
QC	Quorum Certificate
LSO	Leader-Speaks-Once
CHLC	Consecutive Honest Leader Condition
VRF	Verifiable Random Function
TC	TENTATIVATELY-COMMIT messages

1 Introduction

Contents

1.1 Motivation

In this work, we address the problem of implementing permissioned blockchains that can scale to large numbers of participants. Most permissioned blockchains are based on variants of Byzantine Fault Tolerance (BFT) consensus protocols that can offer *finality*, i.e., when a block is decided, it can no longer be reverted. In contrast, most permissionless blockchains prioritize consensus among many participants using computational complexity, sacrificing finality. However, a problem of BFT protocols is that they are difficult to scale because all participants must engage in multiple rounds of data exchange. With large numbers of participants, this can quickly saturate the network or the CPU resources of one or more participants.

It was recently shown that dissemination/aggregation trees could support the data exchange required by blockchain consensus while distributing the load and avoiding bottlenecks [4]. Previous tree-based work, however, is either agnostic to network topology [5, 6] or assumes homogeneous environments [4]. In particular, the Kauri [4] system creates random dissemination trees that are oblivious to the topology and heterogeneity of the network. The tree configuration can poorly impact the system's performance if deployed in a global setting.

However, using trees in either context increases consensus rounds' latency. Therefore, pipelining techniques have been suggested to mitigate the impact of the increased latency in the system throughput. Kauri [4], like many other BFT protocols, is a leader-based protocol that combines these two techniques. Nevertheless, to maximize the benefits of pipelining, it uses a stable leader policy, i.e., the same leader is used for consecutive instances of consensus until this leader is suspected to be faulty. Furthermore, the same tree is used while a given leader is active.

Pipelining is also used in protocols such as Hotstuff [6], which relies on a star topology and starts the next consensus instance optimistically while the previous instance is still running and piggybacking messages from multiple instances on the same network packets.

Several arguments support changing the leader (and, consequently, the tree) between consecutive consensus instances. For instance, in a blockchain setting, the leader can select which transactions are included in a block and, therefore, has the power to censor some specific transactions; the rotation of the leader task among all participants provides censorship resistance. Changing the leader also allows for a better distribution of the load on the leader during the protocol. Nevertheless, changing the leader frequently raises concerns about maintaining the pipeline started by the previous leader.

1.2 Contributions

The challenge addressed in this work is to devise a strategy to rotate the leader (and the associated dissemination/aggregation tree) and, at the same time, enrich Kauri with heuristics that take into account topology and the heterogeneous environment to create topology-aware dissemination and aggregation trees. In detail, the thesis describes:

- a new rotating leader strategy to achieve a fair and non-deterministic sequence of leaders.
- new heuristics to create dissemination and aggregation trees aware of the topology and the environment in which the system is deployed.

1.3 Results

The thesis has produced the following results:

- An implementation of several heuristics to create the trees to integrate within Kauri.
- An evaluation of these heuristics in realistic scenarios comparing to Kauri's random strategy.

1.4 Research History

Part of the work of this thesis related to building topology-aware dissemination trees has been published as:

- H. Teixeira, L. Rodrigues and M. Matos. Arvores de Disseminaçao e Agregaçao Cientes da Topologia para Suportar Consenso Bizantino em Larga Escala. In Inforum23, Porto, September 2023 [7].

1.5 Organization of the Document

The rest of the thesis is organized as follows: Chapter 2 introduces the fundamental concepts relevant to our work, and Chapter 3 provides an overview of the related work; Chapter 4 describes the strategies and heuristics designed and implemented; Chapter 5 describes the results from our experimental evaluation, where we compare the performance of or approach against previous systems that aim at similar goals; finally, Chapter 6 concludes the thesis and provides directions for future work.

2 Background

Contents

In this chapter, we introduce two relevant concepts for our work: blockchain and its categories in Section 2.1 and Byzantine Consensus and its properties and categories in Section 2.2.

2.1 Blockchain

The term *blockchain* describes the type of data storage used in Bitcoin [8] and similar systems. This report will focus on distributed ledgers that leverage the blockchain as their storage technology. A blockchain is a distributed system with the goal of building a decentralized system that provides a trustworthy service in an untrustworthy environment. The decentralized processes run a protocol to reach an agreement on a system state during normal-case operations or when there are arbitrary faults and an attack occurs [9].

A blockchain consists of data sets that are composed of a chain of data blocks where these blocks have multiple transactions. To ensure integrity, each block contains a timestamp, the previous block's hash (called parent), and a nonce (a random number for verifying the hash). This approach makes the entire chain immutable, as changing a block in any position would require updating every subsequent block. Furthermore, if anyone tampered with the data in a block, the newly calculated hash and the hash stored in the following block would differ, requiring rebuilding the chain from that point onward, making the approach tamper-resistant.

Without trusted elements that are aware of all transactions, building a blockchain requires regular synchronization between the different participants [8]. Therefore, consensus is the primary tool for making such a distributed ledger, discussed in detail in the next section.

These blockchain participants can be byzantine (have arbitrary behavior), suffer crash faults (i.e., may stop executing), or honest (perform as expected). Clients interact with the blockchain by sending requests to the system and waiting for a response. When the transactions related to those requests are validated and performed, the processes deliver the result to the client.

Blockchains can be categorized based on their permission model, as discussed below.

2.1.1 Permissionless

Permissionless blockchains are decentralized ledger platforms in which anyone can participate. It functions as an open setting. However, anyone can read and write to the ledger, allowing malicious users to issue requests that try to subvert the system [10]. An adversary could easily create sufficient instances to subvert the existing consensus, generally described as Sybil attacks. To prevent this, mechanisms like Proof-of-Work can be used. A proof-of-work is typically a mathematical puzzle that must be solved to perform an action, preventing a node from acting as multiple nodes without the corresponding CPU capacity. A permissionless blockchain becomes an open system that every participant can join and leave at any moment. This allows the recruitment of numerous participants for blockchain maintenance. However, the approach also has some disadvantages. For instance, proofs-of-work are very expensive from the point of view of energy consumption and limit the system throughput. It is also important to point out that this approach allows processes to collude to solve the PoW faster in a byzantine context, and the blockchain becomes less decentralized [9].

2.1.2 Permissioned

In permissioned blockchains, the participants are known to each other and must be authorized by some trusted entity. Since they are more restrictive, it allows running traditional consensus algorithms more easily. Most permissioned blockchains are based on Byzantine Fault Tolerant State Machine Replication (BFT-SMR) algorithms, which use less energy on processes running a BFT protocol to agree on the transaction order. Compared with permissionless blockchains, it is provably secure and faster performance-wise [9]. These blockchains are widely used, for example, in the finance and banking industry and Supply chain industry where tracing and tracking, sustainability, time-saving, cost-saving, and preventing counterfeiting are the most wanted features [11]. They employ a Byzantine failure model with N replicas in which the system can tolerate $\frac{N-1}{3}$ faults, and the quorum size is $\frac{N+f+1}{2}$.

2.2 Byzantine Consensus

Byzantine consensus allows correct processes to agree on a single outcome even when some participants exhibit arbitrary behaviour [12].

2.2.1 Nakamoto's Consensus

The consensus algorithm presented by Nakamoto in Bitcoin [8] was the first to be used in permissionless blockchains. Here, the participants (called miners) need to solve the mathematical puzzle (mentioned as Proof-of-Work (PoW)) to participate in the consensus. When a miner solves the puzzle, he is rewarded, incentivizing him to participate. In case two processes solve the puzzle simultaneously, it creates a *fork* where there are differences in the local blockchain of certain participants. A *fork* is solved by the *longest chain rule* where, if two or more blocks have been proposed in conflicting branches of the chain, the participants must choose the branch that is believed to have the most work done. There are some disadvantages. The waste of blocks proposed in the conflicting branch, combined with PoW-based membership selection, leads to the waste of mining power and high energy consumption [10].

2.2.2 BFT Consensus

BFT consensus rises in the context of not all systems needing the flexibility of being an open setting. These consensus algorithms are the main focus of the following sections. They are employed in managed environments where participants are known and can be vetted. The consensus algorithms have benefits like low computational costs, low transaction processing times, and high transaction throughput. Furthermore, unlike Nakamoto's Consensus, BFT protocols can provide strong safety guarantees. However, applying classical BFT protocols in blockchains is still technically challenging due to their scalability issues. Also, classical BFT protocols rarely consider fairness among leaders.

A BFT consensus algorithm needs to satisfy the following properties [12]:

Termination: Every correct process eventually decides some value.

Weak validity: If all processes are correct and propose the same value, v, then no correct process decides a value different from v; furthermore, if all processes are correct and some process decides v, then v was proposed by some process.

Strong validity: If all correct processes propose the same value, v, then no correct process decides a value different from v; otherwise, a correct process may only decide a value proposed by some correct process or the special value.

Both validity notions are acceptable. Weak validity allows correct processes to decide on the initial value of a Byzantine process. With strong validity, however, this is only possible if not all correct processes have the same initial value. Either way, validity states that all operations accepted need to be proposed by a client. However, it does not differentiate between an honest and a dishonest client, creating the possibility of leader-driven censorship where the leader does not include the honest client's proposed transactions. In Section 3.2, we will further detail the leader's impact on these systems. They are the ones that determine which transactions are going to be satisfied. So, the concern of censorship is a fundamental matter that goes beyond the validity property of classic consensus. One solution is a rotating leader approach, hoping a leader will eventually be honest and break censorship. Nevertheless, this could be better since the sequence of leaders is well-known, and the adversary can choose to corrupt the current and next-in-line leaders. Therefore, it becomes essential to have a way to randomize the schedule of leaders to create a moving target for the adversary [13].

Integrity: No correct process decides twice.

Agreement: No two correct processes decide differently.

In BFT consensus, there is an all-encompassing view of the processes called the *View*. The processes are placed in the *View* according to their roles. We use the notion of configuration to represent the roles of each process and how they interact within a *View*.

BFT algorithms can be categorized as leader-based or leaderless. In [14], they are categorized in further detail as:

- Efficient leader-based BFT: target performance to achieve high throughput and low latency in fail-free executions. Ensure the normal-case operation is the most efficient possible [5, 6, 15].
- Robust leader-based BFT: target maintaining availability against potential failures instead of achieving impressive performance in normal-case operations. The main concern is fault tolerance [16].
- Leaderless BFT: tackles the single point of failure on the leader-based approaches and amortizes the coordination of work among a group of or all replicas.

We will focus on leader-based BFT in the partially synchronous model. There are two essential domains to consider in this category of BFT: how to propagate information and choose a leader.

How votes on a block are disseminated and aggregated significantly impacts system throughput and bottlenecks. We will further detail the different types of topologies used by BFT systems in Section 3.1.

3 Related Work

Contents

This chapter discusses some important features of consensus protocols used to implement distributed ledgers. We only consider leader-based protocols where, for each consensus instance, one process is responsible for proposing a value (in our case, a block). In all these protocols, the execution of a consensus instance requires multiple rounds of communication (sometimes also called phases). The exact number of phases depends on the details of each algorithm. In our analysis, we cover four different aspects of the operation of these protocols: how processes communicate in each round (described in Section 3.1), how the leader is selected for each instance (detailed in Section 3.2), whether pipelining is used or not (described in Section 3.3) and how does a heterogeneous deployment impact these protocols (described in Section 3.4). In Section 3.5, we present several relevant systems and analyze them according to the previous four aspects. In the end, in Section 3.6, we provide a comparative analysis of the previously discussed systems.

All the protocols covered are targeted for blockchains and are based on the concept of *chaining*, where the validity of a new block needs to be attested against the entire sequential history of the chain. This is done by analyzing structures called Quorum Certificates. A Quorum Certificate (QC) is a collection of signatures from different participants for a block.

3.1 Communication Patterns

This section will describe the communication patterns used by different protocols, their impact on the number of communication rounds, and message complexity. We will present five network topologies used by most of the protocols covered in this report.

3.1.1 All-to-all

This is probably one of the simplest and most common communication patterns to implement a protocol round. When using an all-to-all strategy, processes that need to send messages in a round send these messages directly to all other processes. Using an all-to-all communication pattern has several advantages. First, it offers low latency when the network is not saturated because messages are sent directly from the sender to the recipients without using other processes as mediators. Second, this pattern is also very robust because faulty processes cannot interfere with the communication between two correct parties. Unfortunately, the message complexity of a protocol using an all-to-all communication pattern is very high, namely $O(N^2)$, where N is the number of processes, and it may saturate the network when the number of participants is large.

3.1.2 Star Topology

An alternative to the all-to-all strategy is to use a process, typically the leader, to relay information between other processes. Thus, the network is organized in a logical star, rooted in the leader: when a process wants to send information to another process, it first sends the information to the leader that aggregates the information collected from different processes and, in a subsequent step, forwards the aggregated information to all participants. This all-to-one and one-to-all communication pattern enables message complexity to be reduced from $O(N^2)$ to O(N). This pattern aims to alleviate the network's load at the cost of increased latency.

3.1.3 Tree Topology

The use of a star topology reduces the message complexity of the protocol. However, the leader may still be overwhelmed with the load, as it needs to receive messages from and send messages to all other processes. A way to reduce the load of the leader is to organize the participants in a logical tree that serves both as a dissemination and aggregation tree. Typically, this tree is rooted in the leader. When a process wants to send information to another process, it sends this information to its parent. The parent aggregates information from its children and forwards it upwards. This process is repeated until the information reaches the root. Subsequently, the leader aggregates the information collected from its children and uses the tree to push the aggregated information to all participants.

3.1.4 Random Topology (Gossip)

Another way to disseminate information during a consensus round is through epidemic dissemination, also known as gossip. When a process wants to send information to all other processes, it randomly selects a subset of the addresses and forwards the information to those processes. When a process receives the information for the first time, it repeats this procedure: it randomly selects another subset of addresses and forwards the information to them. The number of processes selected in each step of the gossip dissemination procedure is known as the *fanout*. This technique is quite effective for systems with many participants because a process only needs to communicate with a small subset of the participants, defined by the fanout value. It has been shown that it is possible to achieve reliable dissemination, even in the presence of faults, with fanout values that are logarithmic with the system size. As a result, protocols using gossip typically have a message complexity in the order of $O(N * log(n))$, which is still worse than the linear star-based systems. Although the total number of message exchanges is larger than with star or tree topologies, gossip combines good load distribution with high resilience.

3.1.5 Hierarchical Topology

Lastly, another way of disseminating information is to split the nodes into groups that can be based on geographic proximity, for example. This allows the dissemination to happen within the groups, and then a representative of each group will disseminate to the other representatives. As a result, protocols using hierarchical topologies typically have the message complexity of $O(n^2)$ messages within a group (with n as the group size) and $O(k)$ messages within representatives (with k as the number of groups). Compared to All-to-all communication patterns, this reduces the message complexity but also the resilience guarantees since the failures are limited to the group level instead of regarding the whole system size.

3.1.6 Effects of the Topology on the Protocol

The topology has an impact on the protocol. When an all-to-all pattern is used, processes communicate directly with each other, and correct processes can exchange information in a single communication step. When a star topology is used, at least two communication steps are required to execute a protocol round. With a tree or gossip topology, even more, steps are required to execute a single protocol round. Furthermore, consensus protocols require multiple protocol rounds to terminate. A protocol such as PBFT [5], based on an all-to-all pattern, uses three communication rounds. HotStuff [6], which uses a star topology, requires four communication rounds. The extra round is needed because, when using a star topology, reliable communication between correct processes in a given round is no longer guaranteed (because the root of the star may be faulty and omit messages). As will be discussed in Section 3.2.2, using pipelining techniques can mitigate the latency increase caused by additional protocol rounds and communication steps in each round.

It is important to mention why multiple rounds are needed and their goal. We need at least a round to collect votes on a specific value and another round to propose the value to all processes and acknowledge it. The processes, however, can receive several values to vote. In a byzantine setting, there are several reasons for a process to receive conflicting values. It can be a byzantine leader that sends one value to some processes and another to others, or it can be a byzantine process with arbitrary behavior. Either way, there needs to be extra message exchanging to vote from a quorum of votes. Not only do we need multiple rounds of voting when we reach a decision, this one is unrevokable, and we need to inform the decision to all processes. This results in the three rounds for PBFT [5] and four for Hotstuff [6] mentioned.

3.2 Leader Change

In a blockchain setting, it is required to run a consensus for each block created. Thus, a sequence of multiple (chained) consensus instances needs to be executed. These instances can be executed using the same or different leaders and the same or different topologies. For instance, a protocol could keep the same leader but use a different dissemination tree in different instances, or even use a tree topology in one instance and a star topology in another. We now discuss the challenges related to the choice of the leader for a given instance. Namely, we discuss the different strategies to i) decide if two consecutive instances should be executed by the same leader or not; ii) if the leader changes, how correct participants agree that a new configuration is to be used for the next instance and, finally; iii) how the participants select the next configuration (including the next leader).

3.2.1 When to Change the Leader

There are two main approaches to deciding if a leader should be kept or changed when running two consecutive instances of consensus. Namely:

A – The stable-leader approach. In this approach, the same leader is kept across multiple consecutive instances of consensus until it is suspected of failing.

There are some advantages of keeping the same leader across multiple instances. In the first place, there is generally some cost involved in the process of changing the leader; at minimum, one needs to ensure that the leader, for instance, n, is aware of the outcome of instance $n-1$ (i.e., of the previous block). Keeping the same leader avoids this cost. Also, if the system is operating as expected, this may indicate that the leader is correct and that the current configuration is performing adequately; changing the leader risks deteriorating the system's performance.

There are, however, also some disadvantages to keeping the same leader across multiple instances. In a blockchain setting, the leader selects which transactions are included in a given block, thus the leader has the power to censor requests from clients. This means that a Byzantine or rational leader can drop requests selectively, influencing the fairness of the blockchain for a long time in a stable-leader approach. Also, the stable-leader approach has been shown to be vulnerable to performance attacks [17]. Performance attacks are described as occurring when processes send correct messages slowly but without triggering protocol timeouts. This means these processes are correct in the eyes of the BFT properties but can drastically degrade performance. This effect can be exacerbated by other mechanisms used to tolerate network instability. For instance, many protocols double the timeout values when the network is unstable but have no clear mechanisms to reduce timeouts again, allowing faulty processes to perform even slower without triggering timeouts.

B – The rotating-leader approach. In this approach, replicas take turns being leaders of consecutive consensus rounds. This is a proactive approach since it rotates the leader regardless of whether or not the current one is correct.

There are some advantages to changing leaders for each consensus round. In the first place, it tackles the censorship of client requests. By rotating the leader, we know that eventually, the system will be led by an honest process. Secondly, since the leader is the process that does more work, in terms of having an even distribution of work, this approach allows for a balanced distribution of the leader's extra work.

However, this approach also has some drawbacks. Rotating the leader for each round makes view-change a recurrent procedure that becomes part of the system's critical path. So, depending on the view-change protocol, it will have a different impact on the overall system. Finally, this approach does not avoid bad leaders: if we have a slow or faulty process, it will periodically become a leader.

3.2.2 Leader Change Procedure

The view-change procedure is used by every BFT-related system when there is a view-change, whether the change is periodic or just occurs when there is a fault. This procedure dictates how correct participants must agree that a new configuration will be used for the next instance. The communication pattern impacts how this handoff is done from one view to the other.

A – The all-to-all handoff We will start by detailing the hand-off procedure when an all-to-all communication pattern is used. The procedure is triggered by a timeout in a replica, that causes the replica to stop processing the consensus messages and to initiate the view-change by broadcasting a VIEW-CHANGE message. Eventually, the same trigger also occurs at other replicas, which execute similar steps. After this, the leader of the new view will eventually receive a quorum of valid VIEW-CHANGE messages for the new view. This new leader will compute a NEW-VIEW message with the state that needs to be transported to the new view. This NEW-VIEW message will be broadcasted to all replicas, and the leader will enter the new view. The replicas will accept valid NEW-VIEW messages, broadcast a message for each proposal in the state inside the NEW-VIEW message, and enter the new view. It is important to note that this procedure has quadratic message complexity.

B – The one-to-all handoff Now we will detail the hand-off process in a one-to-all communication pattern. The leader talks with all replicas but the replicas only talk with the leader, an algorithm with linear complexity. In the case of a view change, some changes are made in the first and last phases of the normal-case protocol. The replica enters a new view in the first phase and sends a NEW-VIEW message to the corresponding leader. This leader collects the NEW-VIEW messages and broadcasts the new node as a proposal. This does not change the linear complexity of this phase. Each replica executes the request in the last phase and starts the next view. This allows to maintain the linear complexity of the algorithm.

3.2.3 Selecting the Next Configuration

Every system must have a policy to determine the new view's configuration (including the new leader). Note that for star and all-to-all topologies, selecting the next configuration is simply a matter of selecting the next leader. The selection process can simply follow a predefined sequence of leaders (typically, in a round-robin manner) or be based on the performance of the different processes (obtained via some monitoring technique). In any case, the goal is to select a configuration that allows consensus to terminate, what is known as a *robust* configuration:

Robust Star [4] A star is said to be robust if the leader is correct, and non-robust if the leader is faulty.

The tree topology is special since we must choose the next leader and the whole tree configuration. The goal is to find a robust tree configuration.

Robust Tree [4] 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.

A robust star configuration will eventually be found in at most $f + 1$ steps for previous leader-based protocols. On the other hand, a tree reconfiguration strategy is much more complex since many possible configurations exist. One that considers all possible configurations may require a factorial number of steps [4].

3.3 Pipelining

Consensus protocols require multiple rounds of communication to terminate [18]. In chained consensus, if one starts the $n + 1$ instance only after instance n has terminated, the throughput of the system is limited by the latency of the communication rounds. One way of circumventing this limitation is to start optimistically instance $n+1$ before instance n terminates. In chained consensus is possible if the leader of the instance $n + 1$ knows the value proposed in instance n. This technique is known as *pipelining*. With pipelining, different rounds of multiple consensus instances are executed in parallel (for instance, the first round of instance $n + 1$ is run in parallel with the second round of instance n). Messages from multiple instances can often be piggybacked in a single network packet. The pipelining technique needs to be carefully parameterized according to the system characteristics to be effective.

In a stable-leader approach, pipelining is mostly done like the following. One leader handles the pipeline through multiple views. The leader process starts new instances of consensus while processing the previous rounds. If there is a view change, the pipeline will stop, and the next leader will continue the rounds.

In a rotating-leader approach, one common way of pipelining is with a mechanism called Leader-Speaks-Once (LSO). Each leader proposes and certifies a single block and rotates so that we limit the leader's influence. This means the leader of the instance n will execute its first round of consensus. Then, a new leader will do the first round of the instance $n+1$ with the second round of the instance n. This is repeated for all rounds of consensus. Thus, if a protocol requires four consensus rounds, four leaders will be involved, each coordinating a different round. Also, in the same protocol, a single process may be required to execute four distinct rounds of four distinct instances of consensus in parallel.

However, although this approach is desirable for blockchains, it introduces liveness concerns [19]. One concern is defined as:

Consecutive Honest Leader Condition (CHLC) [19] Chained LSO protocols require the formation of k QCs in consecutive views to commit a block (where $k \in \{2, 3\}$ depending on the protocol).

CHLC can be a significant exploit for a byzantine process. The concern is transversal to most protocols that combine chaining and a rotating-leader approach. For example, applying this protocol in Hotstuff [6], the leader will coordinate one phase and rotate. The chain structure can be used to pipeline the commands and have one QC serving as a QC for each of the Hotstuff phases for different blocks as depicted in Figure 3.1.

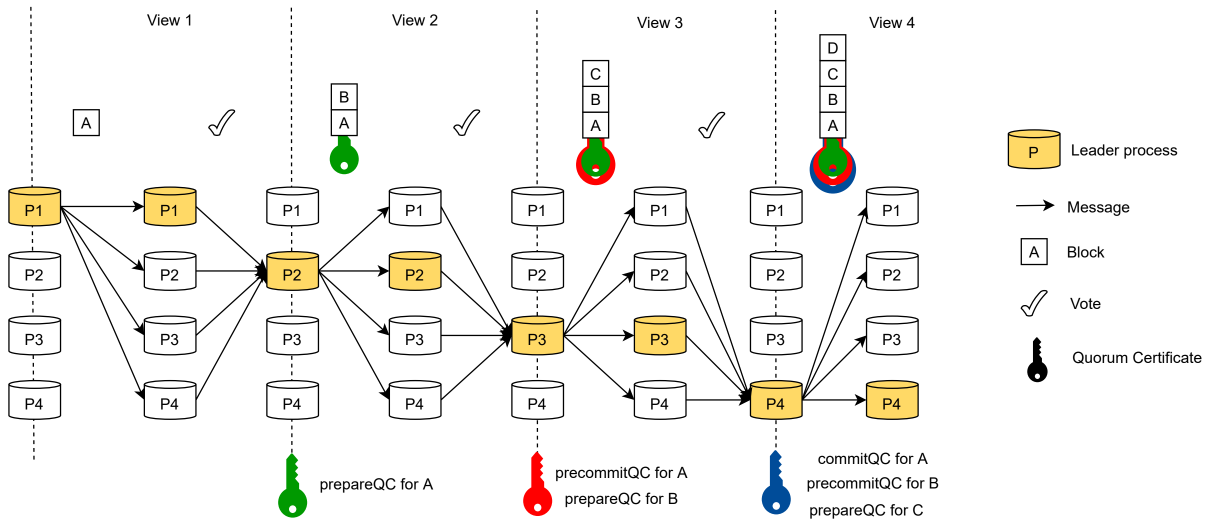


Figure 3.1: Chained LSO Hotstuff

Assuming a baseline protocol needs three voting rounds, one for the proposing stage and two for the committing stage. The first phase mentioned as *Prepare phase* creates a *prepareQC* ensuring agreement on block A. In the second and third voting rounds, a *pre-commitQC* and a *commitQC* are created, respectively, so the processes become locked on a value to decide. Figure 3.1 demonstrates an example of a chained version of the protocol described above trying to commit four blocks: A, B, C, and D. In view four, the QC certifies block A as a *commitQC*, *pre-commitQC* for block B, *prepareQC* for block C and D is proposed. To commit A (the first block proposed), we need three QCs and four consecutive honest leaders. If a failure occurs in view 4, the protocol has to restart the committing process for the block to preserve safety. Thus, for a total number of 4 replicas, a single faulty leader can prevent the protocol from committing any block.

A recent protocol, Siesta [19], proves that CHLC cannot be relaxed for arbitrary faults but proposes a way to relax the condition for the case where we have faults by omission.

3.4 Heterogeneity

The following discussion focuses on the impact of using geo-distributed and heterogeneous networks on the execution of consensus algorithms.

The ability to support the execution of Byzantine consensus algorithms on large-scale networks with a large number of participants is a determining factor for developing highly decentralized blockchains. Many Byzantine consensus algorithms use either an *All-to-All* communication pattern (such as, for example, PBFT [5]) or a *One-to-All* communication pattern (such as, for example, Hotstuff [6]). In both cases, at least one process has to send (and receive) information to (from) all other processes, creating bottlenecks in the system and making it difficult to parallelize the dissemination. To better distribute the load, algorithms like Kauri [4] simultaneously use a tree communication pattern to disseminate and aggregate information where the tree's root is the consensus leader. This tree topology opens new doors to parallelizing dissemination.

3.4.1 The Challenges of Using a Geo-Distributed Network

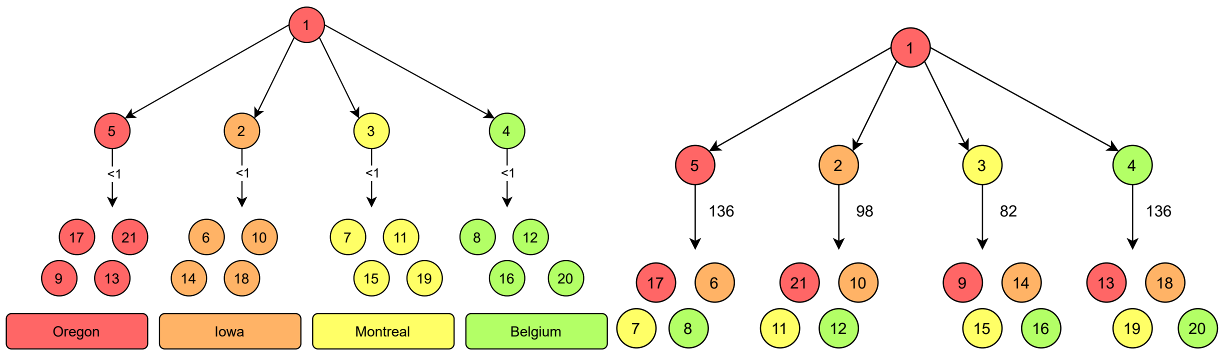
Systems such as ResilientDB [1] have shown that to achieve good performance in geo-distributed networks, it is crucial to avoid crossing high latency links (such as between different data centres) multiple times and to allow the propagation (or aggregation) of information in geographically distinct areas to occur in parallel. Note that the latency between machines in the same data centre or geographic region is significantly lower than between machines in distant regions, as illustrated in Table 3.1. This requires the algorithm to consider the network topology.

The tree communication pattern allows dissemination to be parallelized and, if the configuration is aware (or informed) of the network characteristics, can favour communication between geographically close nodes. Figure 3.2 illustrates an informed configuration's impact on dissemination time. Consider

	Latency (ms)					
	O	I	M	B	T	S
Oregon (O)	$\leqslant 1$	38	65	136	118	161
Iowa (I)		$\leqslant 1$	33	98	153	172
Montreal (M)			$\leqslant 1$	82	186	202
Belgium (B)				$\leqslant 1$	252	272
Taiwan (T)					$\leqslant 1$	137
Sydney (S)						$\leqslant 1$

Table 3.1: Communication latency between different Google data centers as measured by ResilientDB [1].

21 nodes evenly distributed across the first 4 data centres in Table 3.1. A configuration informed by the network, such as the one illustrated in Figure 3.2(a), favours local communication because the second level of the tree does not perform any communication between different data centres, which allows for substantially reduced latency of dissemination/aggregation. On the other hand, a random configuration, as illustrated in Figure 3.2(b), does not have this concern, forcing communication between different data centres at the two levels of the tree.



(a) Configuration informed by the geographical layout of the nodes.

(b) Random configuration.

Figure 3.2: Comparison between a configuration informed by the geographic arrangement of participants and a random configuration. The values next to the arrows indicate the maximum latency associated with that connection.

3.5 Some Relevant Systems

In this section, we describe some relevant systems and frame them in the context of the design space introduced previously. For each system, we will provide an overview of its context, contribution to the state-of-art, advantages and disadvantages, and categorization in terms of communication pattern, leader change protocol, whether pipelining is leveraged in case of a rotating approach and if it leverages heterogeneity in its design.

3.5.1 PBFT.

This PBFT [5] protocol uses an all-to-all communication pattern and a stable-leader approach. It is a practical algorithm that tolerates faults in an asynchronous Byzantine environment. Previous systems assumed a synchronous system and relied on failure detectors. In PBFT the normal-case operation involves three steps of all-to-all communication. The execution of the protocol is depicted in Figure 3.3. When clients make requests, the leader proposes an order for the requests by sending a PRE-PREPARE

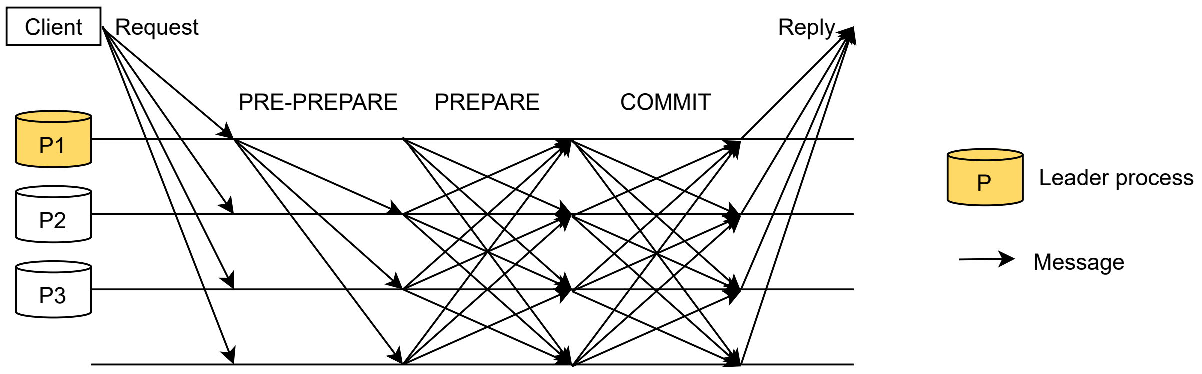


Figure 3.3: Normal-case PBFT protocol

message to the replicas. By receiving this message, the replicas acknowledge the proposal by sending a PREPARE message to each other. After collecting $2f + 1$ matching PREPARE messages, a replica will send a COMMIT message to others. When a replica receives 2f + 1 matching COMMIT messages, she knows that sufficient correct replicas reached a consensus and can execute the operations in the transaction and reply to the client. Since PBFT is a leader-based protocol, replicas must monitor the leader's state to guarantee the system's liveness when the leader fails. Therefore, every replica starts a timer when a request is received. When this timer expires, processes double its value; if the system fails to make progress with the larger timeout value, the view-change protocol is started. Thus, PBFT follows a stable-leader approach.

PBFT [5] is vulnerable to performance attacks when malicious replicas collude to manipulate the value of the timer. The value of this timer doubles during view-change as part of the protocol, and when it becomes large, a faulty leader can deliberately delay some client requests [9]. The quadratic communication complexity in the normal-case protocol is also a very limiting issue regarding system scalability.

3.5.2 HotStuff.

Hotsuff [6] uses a star topology and assumes a rotating-leader approach with a round-robin sequence. It introduces a new goal in BFT algorithms: a system with a linear view change and optimistic responsiveness. Informally, optimistic responsiveness requires that a non-faulty leader, in beneficial circumstances, can drive the protocol to consensus in time depending only on the actual message delays, independent of any known upper bound on message transmission delays. The execution of the protocol is illustrated in Figure 3.4. The leader broadcasts to the replicas and collects votes from them. This process is

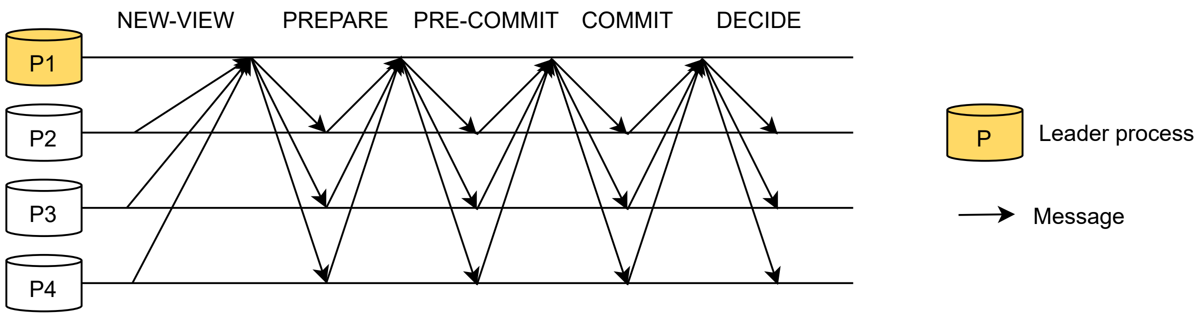


Figure 3.4: Normal-case Hotstuff protocol

repeated in three phases. This one-to-all communication pattern allows the reduction of the message complexity from $O(N^2)$ to $O(N)$. However, compared to PBFT, the number of steps is increased from three to seven. In PBFT, we have a communication step per phase, and now we have two steps per phase since the leader needs to disseminate and aggregate the replicas' votes. Hotstuff uses pipelining techniques to compensate for the extra steps and improve throughput. As a result, it can execute four instances in parallel, thus improving the throughput by four [20]. However, the leader needs to disseminate N messages per round and receive a quorum of votes: this creates a bottleneck resulting in limitations to scalability [4].

3.5.3 Kauri.

Kauri [4] is a consensus protocol that uses a tree topology to perform vote aggregation and disseminate information to address scalability limitations like the leader bottleneck mentioned in Hotstuff [6]. When in a tree topology, each process has to disseminate at most *m* messages and processes at most *m* votes, where *m* is the number of children a node has in a tree (also referenced as the fanout). Thus, the process is busy for a shorter time and has longer idle times.

In Kauri's protocol, the leader broadcasts a block proposal to his children, and the children will do the same if they also have children. Every tree parent will collect the votes from its children until the leader collects a *prepareQC* of signatures for the block. Then, the leader will broadcast a *prepareQC*prepareQC and, in the same way as the phase before, will collect a pre-commitQC, becoming locked on the value proposed. Then, the leader broadcasts a pre-commitQC and collects a *commitQC*. Finally, the value becomes decided, the leader broadcasts a *commitQC*, and the other replicas will verify it and decide.

Kauri follows a stable-leader approach, and the view-change protocol can be seen as a one-to-all handoff even though we have a tree topology. The internal nodes aggregate the children's votes into one single vote, which is O(m), the verification process is O(1), and the worst-case scenario is linear complexity. When there is a faulty leader, Kauri follows the same strategy as Hotstuff, where the replica starts a view-change, the new tree configuration is built, and sends a NEW-VIEW message directly to the next leader. The new leader receives the messages and either continues the previous work or proposes a new one. It still does not surpass linear complexity. The communication pattern is depicted in Figure 3.5

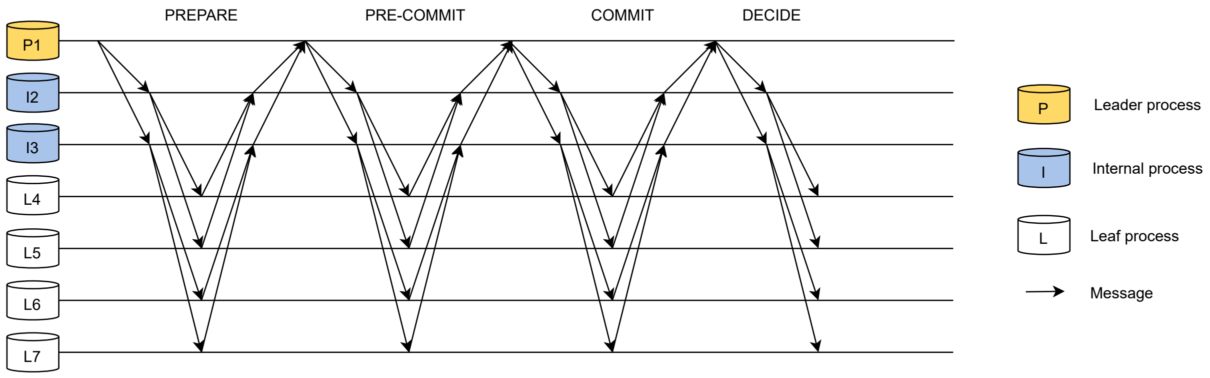


Figure 3.5: Kauri's communication protocol

The main difficulties in tree topologies are the additional latency and the complex reconfiguration strategy. While trees allow a more balanced way to distribute the load among replicas, there is an additional round-trip in the dissemination and aggregation patterns affecting system throughput [4]. Since Kauri builds the trees randomly from a set of processes if we are in a global environment, this allocation can severely impact the system latency.

To mitigate the additional round-trip, Kauri presents a novel pipelining technique. The pipeline stretch is a parameter that tells the system how many instances of consensus can be sent to use the extra time he has because of the tree topology. This technique is depicted in Figure 3.6. This pipeline stretch is based on the following:

A – Sending time. Sending time is the time a process takes to disseminate a block to its children. It is influenced by three main factors: bandwidth, fanout, and block size.

- The smaller the bandwidth b, the longer it takes to disseminate.
- The larger the fanout m and block size B, the longer it takes to disseminate.

In Kauri [4] is represented as $\frac{mB}{b}$.

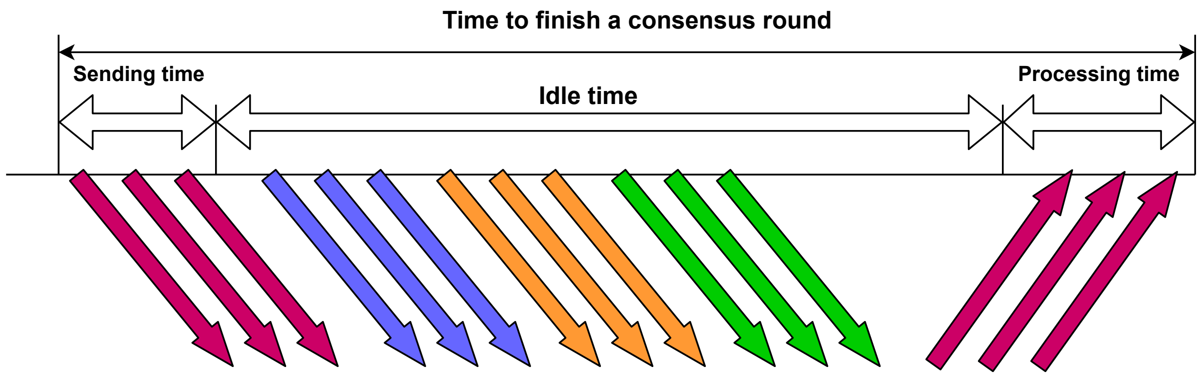


Figure 3.6: Representation for pipeline stretch of four in a consensus round.

B – Processing time. Processing time is the time it takes to aggregate and verify the signatures received from the children. It is influenced by the fanout m and the processing time per signature Φ. Therefore, it is represented as $m * \Phi$ since the larger the fanout and signature processing time, the longer it takes to verify and aggregate.

C – Idle time. The in-between time is called idle time and is what the pipeline stretch leverages.

3.5.4 Byzcoin.

Byzcoin [13] is a solution that, like Kauri [4], leverages a tree topology but employs a rotating-leader approach. It uses a tree communication pattern to optimize transaction commitment and verification during normal-case operations while guaranteeing safety and liveness. Trees are built randomly from a VRF, creating a non-deterministic sequence of configurations. Byzcoin wants to apply trees in Bitcoin [8], which is not a permissioned blockchain. To address this challenge, it creates small groups dynamically from windows of recently mined blocks and performs consensus in each. Furthermore, to maintain the fairness-enforcing benefit of Bitcoin's leader election, Byzcoin makes a PBFT view-change every time the leader signs a proposal. If there is a conflict, another view-change is performed in which the successful miners attempt to persuade other participants to adopt their block. Finally, when a miner creates a block, it forms a tree for collecting signatures with himself as the root. Like many BFT protocols in practice, this protocol is still vulnerable to slowdown or temporary DoS attacks by byzantine nodes, excluding processes from consensus and censoring transactions.

3.5.5 Prosecutor.

Prosecutor [15] has a star topology and follows a stable-leader approach. It intends to penalize Byzantine servers, ensuring they do not usurp leadership over time. When we have a faulty leader, no consensus can be processed, and the system becomes unavailable. This is a problem for all leader-based BFT consensus algorithms. The Prosecutor's dynamic-penalization election technique mitigates this problem by reducing the time the system is unavailable. The star topology allows the achievement of linear message complexity. It creates a novel technique to penalize Byzantine servers allowing replicas to actively claim to be the new primary and apply computation penalties (PoW) on candidates to primary. The more failures a candidate has exhibited, the greater the computation penalty. This way, it entices replicas to operate correctly to avoid performing computation work.

However, PoW-like penalization may become less efficient if faulty servers have a strong computation capability. In this case, Prosecutor may suffer a long period without a correct leader before Byzantine servers exhaust their computation capability [14].

3.5.6 Spin.

Spin [16] has an all-to-all communication pattern and follows a rotating-leader approach. The spinning algorithm tries to deal with PBFT stable leadership vulnerability to performance attacks previously mentioned [14].

Its normal-case operation is similar to PBFT's but loses the PBFT's view-change protocol since it changes every time. This new behavior has an apparent problem: if the leader is constantly changing, then a faulty server will always periodically be the leader. For this, it uses a blacklisting technique to prevent *f* faulty replicas from becoming a leader when they exhibit faulty behavior. A merge operation substitutes the PBFT's view-change protocol, where the algorithms merge the information from different replicas to agree on the accepted requests and can go into the next view.

Nevertheless, when a failure is in a replica and not in a leader, it still incurs extra message-passing, impacting the system performance [14].

In terms of pipeline, the basic spinning algorithm can be generalized to run a pre-configured maximum number of consensus rounds called *window size (w)*. When a request is received, if the leader did not already launch *w* consensus rounds, it starts a new one. If not, the request is kept for the next view.

3.5.7 Carousel.

Carousel [21] focuses on the main drawback of the round-robin approach followed in Hotstuff [6]: it is not bounded to the number of faulty replicas that become leaders impacting latency negatively even in crash-only executions. Therefore, it proposes a new leader-rotation mechanism applied in a Hotstuff-based system. It is considered the number of faulty leaders in crash-only executions after the global stabilization time (GST), a property called Leader-utilization. The main idea is to track the involved parties via the records of their participation (for example, signatures) in the chain and elect leaders among them. In other words, allow the use of the reputation of a process to avoid crashed leaders in crash-only executions. At the beginning of a round, each honest party checks if there is a committed block B at round *r - 1*. From this, we can guarantee that the endorsers of B, at least in round *r - 1*, did not crash, adding them to the set of possible leaders. After that, we exclude the f latest authors of committed blocks from the set for chain-quality purposes. Then, finally, the leader is chosen from the remaining. This paper shows that latency and throughput improved in the presence of crash faults compared to the Hotstuff approach with a round-robin election mechanism.

3.5.8 Gosig.

Most protocols mentioned above allow adversaries to launch DDoS attacks on honest processes and partition them from others. Moreover, with the leader as a bottleneck, the system can become limited by the slowest process in the propagation process due to participants' different capacities.

In Gosig [22], the consensus protocol operates in rounds and appends one block to the blockchain per round. Each round has a proposing phase. In this phase, the proposers are selected, and each creates a block and broadcasts it to all processes. Next is the signature collection stage, where each process chooses a block he has received to vote on, signs his decision, and relays the aggregated signatures. Under this consensus protocol, a gossip network is used to propagate all messages. Furthermore, it uses a rotating-leader approach since, for each new block, a random proposer is elected using a Verifiable Random Function (VRF). The protocol is depicted in Figure 3.7 At the start of each

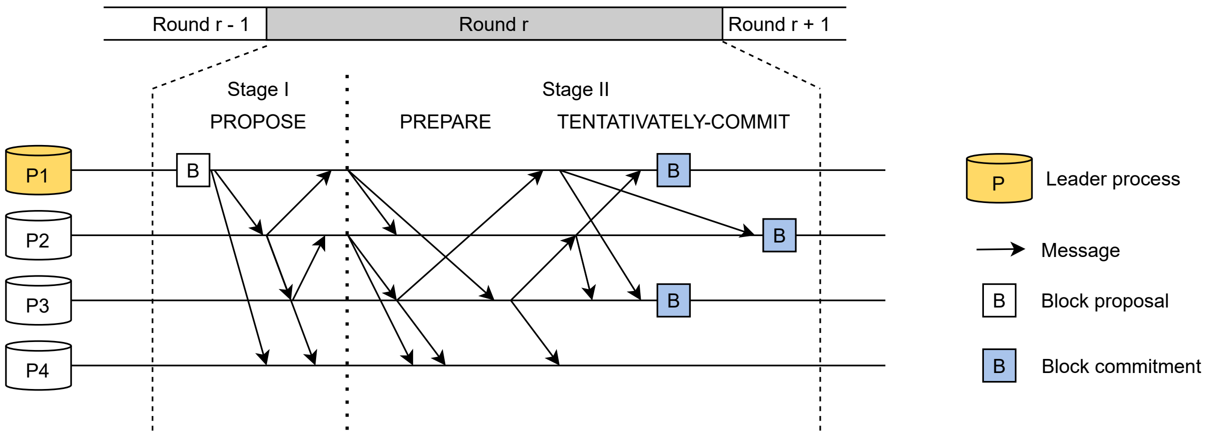


Figure 3.7: A round of consensus with Gosig's protocol.

round, some processes secretly become potential proposers to keep the leadership sequence hidden from attackers. In the first stage, the proposers disseminate the new block in the gossip network. The next stage is to agree on a block proposal by signing it and gossiping a PREPARE message with it. After receiving $2f + 1$ signatures, the process tries to commit the block by sending TENTATIVATELY-COMMIT messages (TC). Finally, the process commits the block when it receives $2f + 1$ TC messages.

Using a VRF to elect the leader, creates a non-deterministic leadership sequence. The goal is to minimize the attacks on the elected leader. Since the leader is only known at the time of the consensus round, and it is not a predefined sequence that the attacker knows, the system gains high censorship resistance. In terms of pipeline, Gosig uses a technique called *transmission pipelining* that considers the communication pattern of the BFT protocol and the gossip network.

3.5.9 Algorand.

Reaching a consensus in an open setting is hard because anyone can participate, and an attacker can create several pseudonyms (Sybils), making the traditional consensus impossible to apply. Other protocols use PoW like Bitcoin [8], but Algorand [23] uses a Byzantine agreement protocol. Algorand wants to avoid Sybil attacks, scale to millions of users, and be resilient to DoS attacks. To address these goals, he prevents Sybil attacks by assigning weights to each participant and guaranteeing some fraction of the weight is from honest users. In addition, Algorand achieves scalability by choosing a committee to run each step of the protocol. To prevent an attacker from targeting committee members, he selects the committee in a non-interactive way by using VRF; the attacker will only know which participant to target once he is already participating in the consensus. This protocol is implemented over a gossip network, and the committee members can be replaced after every step to mitigate attacks on them following a rotating-leader approach. However, a committee solution has the drawback of affecting the system's resilience.

3.5.10 GeoBFT.

In the realm of blockchain applications, the use of Byzantine fault-tolerant consensus protocols is essential to maintain a consistent state among all replicas, even in the presence of faults or malicious participants. However, these traditional protocols [5,6,24] are not equipped to handle consensus among a large number of replicas distributed across a vast geographical area. They were designed for more limited deployments and thus did not account for this scenario. The Geo-Scale Byzantine Fault Tolerant consensus protocol, GeoBFT [1], is designed for excellent scalability by using a topological-aware grouping of replicas in local clusters, introducing parallelization of consensus at the local level, and minimising communication between clusters. In GeoBFT, the replicas are grouped into clusters by region, and each cluster makes consensus decisions independently. Each cluster locally replicates a transaction using PBFT at the beginning of each round. Next, each cluster shares the locally replicated transaction log with all other clusters through a novel global sharing protocol that optimistically performs minimal inter-cluster communication. Finally, after receiving all transactions, each replica can order them and execute them. This process is depicted in Figure 3.8.

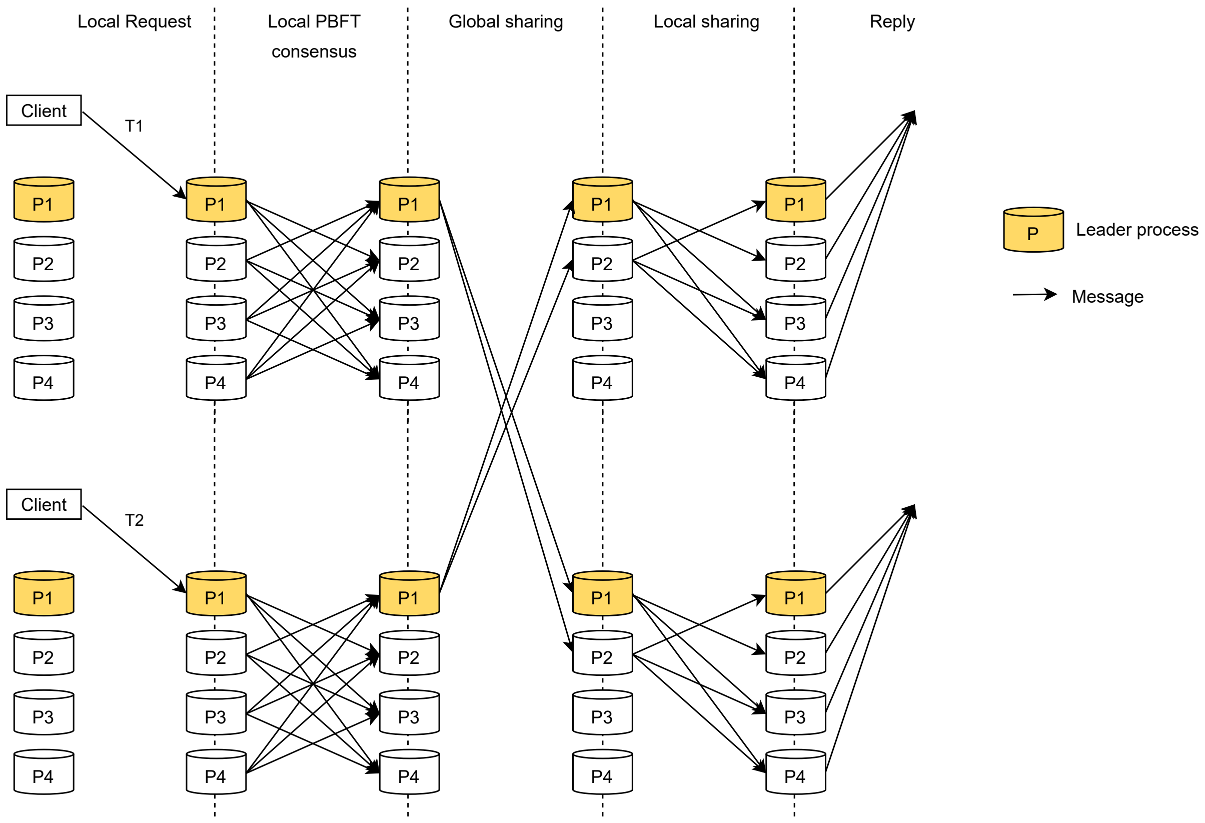


Figure 3.8: Representation of the normal-case algorithm of GeoBFT running on two clusters.

GeoBFT's new communication pattern becomes highly scalable when enabling geo-scale deployments. But needs total replication, forcing communicating large messages among geographically distant replicas.

3.5.11 Other systems

The protocols mentioned above use three or four rounds to reach a consensus, but some solutions focus on reducing this number of phases.

There have been several attempts to reduce the number of phases Hotstuff [6] needs to commit. However, these approaches always have a trade-off: a more expensive view-change for a two-phase normal case. Marlin [25] is an example that achieves consensus in two phases in the normal case and has at most three phases in view-change executions. Zyzzyva [24] introduces a linear path into PBFT. Thus, this optimistic path has linear complexity, while the leader replacement protocol remains $O(N^3)$.

On the other hand, the authors of Aardvark [26] argue that a limited normal-case performance reduction is preferable to the existing loss in availability on the view-change execution.

Aardvark [26] uses monitoring solutions to decide when to change leaders. It leverages PBFT view-change protocol, but the correct primary does not dominate forever. Instead, it imposes two expectations: the primary has to be sufficiently and timely issuing PRE-PREPARE messages; maintain sustained throughput. The replicas monitor the leader; they initiate a view change if he fails these expectations. It still leaves uncertainty since correct leaders can be mistakenly replaced with the system, making no process and imposing unnecessary view-changes [14]. Still, the authors reinforce that the benefits of evicting a faulty leader outweigh the recurring costs of performing view-change regularly.

3.6 Discussion

Table 3.2 provides a comparison of systems covered in Section 3.5. In our analysis, we focus on six key factors. The first two factors capture the communication pattern used to execute rounds and the strategy used to change leaders. Next, for the *Non-deterministic sequence* factor, we focus on whether systems use a deterministic sequence of leaders; systems that use a deterministic sequence of leaders may be more vulnerable to attacks, given that attackers can exploit this knowledge to their benefit. The *Leverages pipelining* factor captures if the systems use pipelining techniques to improve throughput. The following factor is regarding load balancing. We consider two definitions of load balancing in this discussion: intra-round load balancing (the load is distributed within a round) and cross-round load balancing (the load is distributed in the overall execution across multiple rounds). Regarding load balancing on nodes, Kauri [4] offers intra-round load balancing by distributing the load in a tree topology. On the other hand, various systems like PBFT [5] and Spin [16] provide cross-round load balancing. Here, the

	Topology	Leader election approach	Non-deterministic sequence	Leverages pipelining	Load balancing on nodes	Aware of network conditions
PBFT [5]	Clique	stable	✗	✗	✗	✗
Spin [16]	Clique	rotating	✗	✓	✓	✗
Prosecutor [15]	Star	stable	✗	✗	✗	✗
Hotstuff [6]	Star	rotating	✗	✓	✓	✗
Carousel [21]	Star	rotating	✗	✗	✗	✗
Gosig [22]	Random	rotating	✓	✓	✗	✗
Algorand [23]	Random	rotating	✓	✗	✓	✗
Kauri [4]	Tree	stable	✗	✓	✗	✗
Byzcoin [27]	Tree	rotating	✓	✗	✓	✗
GeoBFT [1]	Hierarchical	stable	✗	✗	✗	✓
Kauri Adaptive	**Tree**	**rotating**	✓	✓	✓	✓

Table 3.2: Comparison of existing algorithms.

leader's load is distributed per all processes within several rounds. Therefore, in the *Load balancing on nodes* factor, we focus on the cross-round definition, whether the system is able to distribute the computational and bandwidth usage among different nodes across rounds. The last factor indicates whether knowledge regarding the network conditions is used to achieve better performance.

We described two systems with a clique topology: PBFT [5] and Spin [16]. PBFT still has some issues that can be improved, like the number of resources needed, the vulnerability to performance attacks mentioned in Section A –, and the significant message complexity of the protocol. Spin tries to solve the vulnerability to performance attacks. It uses a rotating strategy with a blacklisting technique for the processes suspected to be faulty. This technique allows for a rotating approach but without a faulty process periodically becoming the leader. Other solutions like Aardvark [26] also solves problems related to attacks on performance. However, its solution to prevent faulty leaders from delaying the service is less efficient than Spinning. Aardvark also changes the leader when it suspects faulty behavior by running a view change operation. However, Spinning does not incur the cost of running a distributed algorithm with several communication steps. As a result, Spin becomes more efficient than Aardvark. In addition, it allows for load balancing among the processes across rounds.

For star topology, we described three more systems: Hotstuff [6], Prosecutor [15], and Carousel [21]. Hotstuff primary advantage is simplicity, enabling pipelining techniques and building large-scale replication services. Some disadvantages are the bottleneck on the leader and the leader election being performed by following a round-robin sequence. A faulty process will periodically be the leader. Also, this round-robin sequence is deterministic and predictable, making the system vulnerable to attacks on the leader. Prosecutor focuses on the impact of having a faulty leader periodically and uses a dynamic-penalization election technique to mitigate this problem, reducing the time the system is unavailable. However, it imposes computational costs on faulty servers, which is a different approach from others since the previous approaches on penalization allow faulty servers to be freely released after pretending to be correct servers. On the other hand, PoW-like penalization may become less efficient if faulty servers have a strong computation capability. Carousel also presents a new technique to deal with the impact of the round-robin approach. It focuses on minimizing the effect of crash-only executions by a participation tracking technique. This new leader-rotation mechanism demonstrated drastic performance improvements in throughput and latency compared with Hotstuff-based systems with a round-robin mechanism.

In the category of Random topology, we mentioned two systems: Gosig [22] and Algorand [23]. These two systems provide solutions to the vulnerability of attacks on the leader. Gosig focuses on minimizing attacks on the leader using VRF so attackers do not know the sequence. Using this method means that the leader is only known when the consensus round starts, so the system tolerates attacks on the leader. Algorand selects consensus groups, using a VRF to prevent attacks targeting the leaders. However, this system has the drawback of using a committee approach for consensus, which limits the system's resilience.

Categorized as a tree topology, we analyzed two existing systems: Byzcoin [27] and Kauri [4]. Both systems use a tree communication pattern to distribute the computational and bandwidth load among processes intra-round. Byzcoin builds trees, using a VRF, to collect signatures when a miner creates a block. However, the throughput depends on the round latency and does not provide a quick recovery. The main advantages of Kauri are load balancing for scalability and a quick recovery strategy. In addition, using a novel pipelining technique, Kauri achieves high throughput as the system grows. However, Kauri uses a stable-leader approach, creating fairness concerns.

Lastly, with a hierarchical topology, we analyzed GeoBFT [1] that provides a topological-aware grouping of replicas in local clusters, introducing parallelization of consensus at the local level and minimising communication between clusters in order to achieve better performance when the system is distributed globally.

We propose Kauri Adaptive, a system with Kauri as the baseline, achieving intra-round load balancing on the nodes and scalability while avoiding throughput limitations due to additional round latency of the tree topology. In addition, we introduce a rotating-leader approach in Kauri to guarantee fairness, a new non-deterministic sequence approach for the tree configurations to limit attacks on the leader, and a new technique to achieve load balancing on the nodes across rounds. Table 3.2 shows that for every topology, there is at least a system that follows a rotating-leader approach and leverages a pipelining technique. There is no version of a system with a tree topology, following a rotating-leader approach, that leverages pipeline and is aware of the network conditions. Thus, we propose a new rotating-leader approach that will leverage network conditions to achieve better performance in global settings.

Summary

In this chapter, we introduced the most relevant approaches and systems to achieve a scalable and resilient BFT consensus algorithm. We discussed their advantages and disadvantages, more precisely, their communication patterns, their leader change mechanisms and their deployments. In the next chapter, we will discuss our contributions. Starting from its model and assumptions, its architecture and implementation.

4

Kauri Adaptive

Contents

In this section, we detail our solution. Our goal is to implement a non-deterministic rotating-leader strategy in Kauri to guarantee load balancing while maintaining the good performance results of the system by leveraging the environment in which the system is deployed. We intend for our new tree rotation algorithm to allow each participant to be a leader only once. This rotating-leader approach results in fairness and censorship resistance in a blockchain.

4.1 Model and Assumptions.

As a model, we assume the same assumptions as Kauri, which are the standard for this type of system [4–6]. In particular, we assume a system consisting of N processes, f of which may be Byzantine subject to the constraint $f < N/3$. Byzantine processes can behave arbitrarily but do not have sufficient computational power to subvert the cryptographic primitives. Additionally, we assume the existence of a system capable of providing the distances, in terms of latency, between the different processes in the system. In practice, this can be achieved with coordinate systems such as Newton [28].

4.2 Architecture

We divided our approach into two distinct problems: i) generate all trees (described as the *Generation phase*); ii) given the set of generated trees, define their sequence (described as the *Sequence phase*).

Next, we will describe each one in more detail.

4.2.1 Generation Phase

In the *Generation phase*, our goal is to generate trees that meet the following requirements: (i) each tree generated is aware of its topology; and (ii) each generation considers the heterogeneity of the environment.

Our approach to generating a topology-aware tree starts from two observations: i) the Internet presents characteristics of a small-world network [29] and ii) nodes participating in permissioned blockchain systems are typically clustered in geographically dispersed data centers [1] that have internally very low latencies and between them substantially higher latencies as illustrated in Table 3.1. Thus, the intuition of our approach is that at the beginning of the dissemination, we spread the information over geographically distant areas and, therefore, potentially using links with higher latency, and from then on, take advantage of the locality and do the rest of the dissemination using local links that typically have lower latencies. The detailed implementation of these algorithms is specified in Section 4.3. The decision process on which heuristic to use is as follows: Can I manipulate the environment in terms of the distribution of nodes through the data centres?

1. If not, we use a Diversity-Aware group distribution to generate the groups and, for each group, create trees with a Generic tree configuration;

2. If we can, we use a Bandwidth-aware group distribution to generate the groups and create trees with a tree configuration that focuses on local dissemination for each group.

This will result in N trees generated.

4.2.2 Sequence Phase

In the *Sequence phase*, we focus on creating the sequence of the generated trees. Our goal is to create a strategy that meets the following requirements: (i) the trees have different roots, meaning that after all the generations, the extra load imposed on the root is balanced by all nodes; (ii) leverages the pipeline by guaranteeing that the leader in configuration *i* was an internal node in the configuration *i - 1*; and (iii) the sequence is not predictable for an attacker to exploit the system.

From the N-generated trees, our approach leverages solutions that use VRF like Algorand [23] and uses them to choose arbitrarily between the direct children of the previous root, as exemplified in Algorithm 4.1. This way, we (i) have a non-deterministic sequence; (ii) assuming the function is uniform in

Algorithm 4.1: VRF logic to add in Kauri

```
Input: $T$ - set of trees generated for each node. $r$ - round.
Output: Next tree $T[k]$.
if In round $r-1$ then
    for $k \in currentTree.children$ do
        $vrfOutput_k, proof_k \leftarrow VRF(SK, seed)$;
if goToNextRound then
    $max \leftarrow MINHASH$;
    $nextTreeLeader \leftarrow \emptyset$;
    for $k \in currentTree.children$ do
        $hash \leftarrow HASH(vrfOutput_k)$;
        if $max \leq hash$ then
            $max \leftarrow hash$;
            $nextTreeLeader \leftarrow k$;
    return $T[nextTreeLeader]$
```

the long term, we guarantee fairness; and (iii) maintain the pipeline by choosing from direct children of the previous root.

4.3 Implementation

In this section, we present the implementation of the heuristics developed to generate Topology-Aware Trees. We start with two heuristics for allocating nodes to groups: (a) the Diversity-Aware distribution and (b) the Bandwidth-Aware distribution. Next, we present three heuristics for selecting the tree's root in case we want to know which root is best for a certain group: (a) the average-latency heuristic, (b) the quorum-driven heuristic and (c) the best-bandwidth heuristic Finally, we show the implementation of the algorithm for building trees from a group in two scenarios: (a) a generic one, where we do not have control of the environment and want to generate the best possible tree for that environment; and (b) a heuristic using local-dissemination, where we have control of the environment and can manipulate it to a specific high-performing setup.

4.3.1 Allocating Nodes to Groups

Since Kauri takes a fixed set of groups as its starting point, the first challenge is determining how to allocate nodes to groups. To do this, we start by introducing the concept of δ-Aglomeration:

δ-Aglomeration A δ-Aglomeration is a set of nodes that are at most δ latency units apart.

Starting from the information provided by the geographic coordinate system and the δ specified by the administrator, we use a clustering [30] algorithm to divide the nodes into several δ-agglomerates. In practice, this roughly corresponds to dividing the nodes by the data centres or geographic proximity in which they are located.

Next, we present two heuristics to distribute the nodes within the groups.

4.3.1.A Diversity-aware distribution

In this heuristic, we create t groups initially empty, which correspond to the groups used by Kauri to define the internal nodes of the tree (lines 1– 2 of the Algorithm 4.2). Starting from the δ-agglomerates previously constructed, we distribute the nodes of each δ-agglomerate across the B_i-groups in a rotating manner as illustrated in Figure 4.1 and defined in the lines 3– 5 of the Algorithm 4.2.

Thus, by spreading the nodes of each δ-agglomerate across the different groups, we maximize the diversity of each B_i-group. This fact will be exploited in the next steps of building a tree from a group, like in the heuristics 4.6.

4.3.1.B Bandwidth-aware distribution

This heuristic depends on having the bandwidth measures provided by the geographic coordinate system for each δ-aglomerate and the block size in the system. With this, we calculate the number of

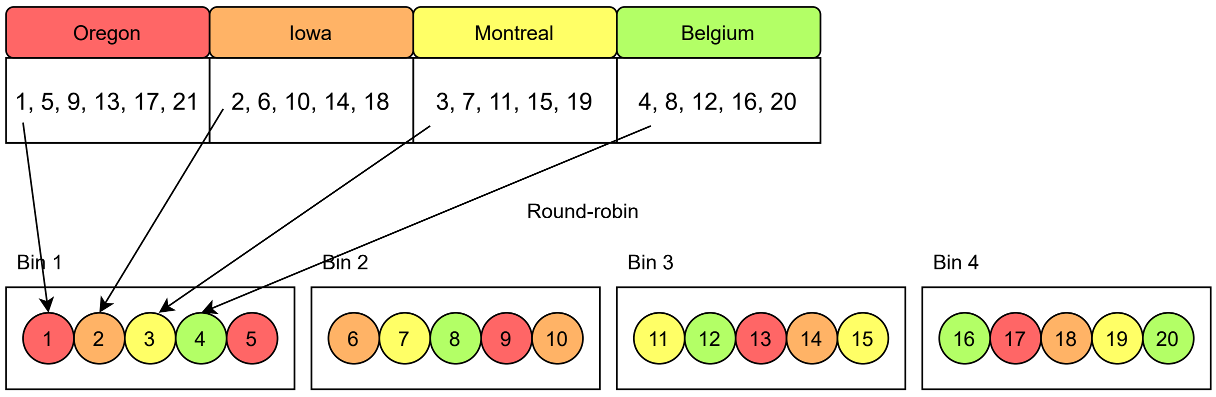


Figure 4.1: distribution of groups based on geographic distribution.

Algorithm 4.2: Alocation of nodes to groups

```
1 for $i = 1, 2, \ldots (N/I)$ do
2     $B_i \leftarrow \emptyset$;
3 for $i = 1, 2, \ldots, I$ do
4     $n \leftarrow$ node from $\delta$-aglomerate not allocated yet ;
5     $B_i \leftarrow B_i \cup \{n\}$ ;                /* Group i with I nodes created */
```

internal nodes per δ-aglomerate i as follows:

$$I_i = \frac{bandwidth_i}{blocksize} + 1$$

Resulting in the number of internal nodes in a group, I, to be the sum of the calculated factors for each δ-aglomerate i. We create t groups initially empty, which correspond, again, to the groups used by Kauri to define the internal nodes of the tree (lines 2– 2 of the Algorithm 4.3). From the number of internals per δ-aglomerate calculated previously, we add that amount of internal nodes from each cluster to each group (lines 3– 6 of the Algorithm 4.3).

Algorithm 4.3: Alocation of nodes to groups

```
1 for $i = 1, 2, \ldots (N/I)$ do
2     $B_i \leftarrow \emptyset$;
3 for $i = 1, 2, \ldots, I$ do
4     for $i = 1, 2 \in C$ do
5         $internals \leftarrow I_i$ nodes from cluster $i$;
6         $B_i \leftarrow B_i \cup internals$;        /* Group i with I nodes created */
```

Thus, we can minimize the latency within each local cluster by creating groups aware of the bandwidth bottlenecks. This fact will be exploited in the next steps of building a tree from a group, like in heuristics 4.7.

4.3.2 Root selection

The selection of the root is fundamental for the system's good performance because the root acts as a leader who, in turn, is responsible for initiating each of the phases of the algorithm. Knowing which root will perform best within a group can be helpful. One possible approach would be for each node in the group to test all possible combinations of trees, considering this node as the root. Despite resulting in the ideal tree, this approach is impractical since the number of combinations to explore is exponential. To design our heuristics, we followed the latency and bandwidth matrix described in Table 4.1 from ResilientDb [1]. Next, we present three heuristics to choose a root from a group.

	Ping round-trip times (ms)						Bandwidth (Mbit/s)					
	O	I	M	B	T	S	O	I	M	B	T	S
Oregon (O)	$\leqslant 1$	38	65	136	118	161	7998	669	371	194	188	136
Iowa (I)		$\leqslant 1$	33	98	153	172		10004	752	243	144	120
Montreal (M)			$\leqslant 1$	82	186	202			7977	283	111	102
Belgium (B)				$\leqslant 1$	252	272				9728	79	66
Taiwan (T)					$\leqslant 1$	137					7998	160
Sydney (S)						$\leqslant 1$						7977

Table 4.1: Communication latency and bandwidth between different Google data centers as measured by ResilientDB [1].

4.3.2.A Average-Latency

In this case, the root will be one of the nodes in the cluster that belongs to the δ-agglomerate with the lowest average latency to the other δ-agglomerates given by the Algorithm 4.4.

Algorithm 4.4: Lowest average latency

```
1 $averageLatencies \leftarrow \emptyset$ ;
2 for $cluster \in clusters$ do
3     for $l \in latencyModel[cluster]$ do
          /* Sum of latencies from the cluster to the others */
4         $latencies += l$
5     $averageLatencies[cluster] = \frac{latencies}{C}$;
6 $centerDatacenter = \min(averageLatencies, key = averageLatencies.get)$
```

A deterministic strategy should be used in case of a tie, such as choosing the δ-aglomerate with the node with the smallest identifier among the tied ones. The intuition is that the root should communicate with all zones to start the dissemination process, minimizing the latency required for this global dissemination. Thus, we want to ensure that all zones start the local dissemination process from the first level at the lowest possible cost.

4.3.2.B Quorum-Driven

The idea behind this heuristic is similar to the previous one. Still, it focuses on the fact that in a blockchain setting, we only need a quorum of answers to proceed with the consensus algorithm. We only need to know the δ-aglomerate with the best K-average latencies. With k as:

$$K = I - \frac{I-1}{3}$$

Thus, we want to try to continue the consensus progress after only waiting for $\frac{2}{3}$ of the answers and choose the root accordingly.

4.3.2.C Bandwidth-Driven

In this case, the root will be one of the nodes in the cluster that belongs to the δ-agglomerate with the biggest average bandwidth to the other δ-agglomerates given by the Algorithm 4.5.

Algorithm 4.5: Biggest average Bandwidth

```
1 $averageBandwidths \leftarrow \emptyset$ ;
2 for $cluster \in clusters$ do
3     for $l \in bandwidthModel[cluster]$ do
          /* Sum of bandwidths from the cluster to the others    */
4         $bandwidths{+}= l$
5     $averageBandwidths[cluster] = \frac{bandwidths}{C}$;
6 $centerDatacenter = \min(averageBandwidths, key = averageBandwidths.get)$
```

A deterministic strategy should be used in case of a tie, such as choosing the δ-aglomerate with the node with the smallest identifier among the tied ones. The intuition is that, since the root should communicate with all zones to start the dissemination process, we want to choose δ-aglomerate, which will less likely create a bandwidth bottleneck for this global dissemination.

4.3.3 Building a Tree from a Group

Given a group derived from any of the previous strategies, we present how to build a tree from it. We implemented two approaches, and both are divided into two sub-problems: (a) configuring the internal nodes and (b) allocating the leaves.

4.3.3.A Generic Heuristic

In this section, we assume that the system administrator defines the number of processes N, the fanout of the tree, and the δ factor detailed above.

This logic is more formally defined in Algorithm 4.6.

Algorithm 4.6: Build an informed tree

```
Input: A list $B_i$ with the group of $I$ internal nodes. $M$ - arity of the tree. A list $remaining$ with the
       leaves of the tree.
Output: Tree $T$ represented as list.
/* Add root to tree                                                    */
1  $root \leftarrow rootHeuristic()$;
2  $T \leftarrow T \cup \{raiz\}$
/* Global dissemination on the first level of the tree                 */
3  for $internal = 1, 2, \dots M$ do
4      $T \leftarrow T \cup$ {node from a $\delta$-aglomerate not yet chosen from group $B_i$}
/* Number of levels                                                    */
5  $L \leftarrow round(log_M(N+1)) - 1$
/* In case there is more than one level of internals                   */
6  $parents \leftarrow$ deeper nodes in $T$ while $L! = 1$ do
7      for $parent \in parents$ do
8          Order remaining internal nodes of $B_i$ per latency to $parent$;
9          $T \leftarrow M$ nodes with less latency;
10     $L \leftarrow L - 1$;
/* Allocate leafs to the deeper nodes                                  */
11 $parents \leftarrow$ deeper nodes in $T$
12 for $parent \in parents$ do
13     Order nodes of $remaining$ per latency to the $parent$;
14     $T \leftarrow M$ nodes with less latency;
15 return $T$
```

A – Configuration of internal nodes. Having selected the root, the next step is to define the configuration of the remaining nodes in the B_i group that will be the internal nodes of the tree. It should be noted that the choice of the root is essential, as described above, and the order in which the internal nodes appear in the tree is also essential to achieve good performance. Specifically, dissemination in Kauri happens, by convention, from left to right. Therefore, the leftmost nodes should be those that have a lower latency connection to the root (lines 3–4 in the Algorithm 4.6). This is relevant because the consensus algorithm does not need to wait for all nodes to make progress. A quorum is sufficient. Thus, by placing the nodes with the least latency for the leftmost root, we maximize the probability of not waiting for a response from the slower nodes further to the right. If the tree has more than two levels of internal nodes (i.e. the root and direct children), we switch the approach to prioritizing local communication. In this case, internal nodes are assigned to parents that belong to the same δ-agglomerate or, if this is not possible, to the parent with the lowest latency (lines 6– 10 in the Algorithm 4.6).

B – Leaf allocation. Once the configuration of the internal nodes is defined, it remains to allocate the nodes of the remaining groups as leaves of the tree. At this point, we want to maximize local communication, so we allocate the leaf nodes of each group to a parent node that belongs to the same δ-agglomerate or, when this is not possible to the parent that minimizes latency (lines 12– 14 in the Algorithm 4.6).

4.3.3.B Heuristic Using Local-Dissemination

The goal is to input restrictions on the environment to achieve the most high-performing tree for this condition. The idea comes from creating subtrees of local dissemination only. Global communication only happens when the roots of each subtree communicate. Having minimized the latency needed by prioritizing the local dissemination, the shape of these subtrees becomes dependent on the bandwidth available. Here, we need the number of internal nodes and how many are from each cluster. The logic from the heuristic Bandwidth-aware distribution can calculate this.

This logic is more formally defined in Algorithm 4.7

A – Configuration of internal nodes. From a group previously defined by the according heuristic previously defined and fixed message size bk, we calculate the maximum fanout wanted for each δ-aglomerate i as so:

$$M_i = \frac{bandwidth_i}{blocksize}$$

This way, for a root of δ-aglomerate i, we configure the internal nodes as follows:

- get one internal node from all other δ-aglomerates and add it as a child (lines 3– 4 in the Algorithm 4.7).
- get the internal nodes from the δ-aglomerate i (the root one) and create a subtree of maximum fanout M_i from these nodes (lines 6– 10 in the Algorithm 4.7).
- for each child of the root that is not from the same δ-aglomerate do the same logic and create a subtree for each with the corresponding maximum fanout M_i(lines 10– 16 in the Algorithm 4.7).

This approach mitigates the bandwidth bottleneck when disseminating within a δ-aglomerate.

B – Leaf allocation. For the leaves, we keep the same goal and go to the deepest internals of each subtree and add those, always considering the maximum fanout for the δ-aglomerate the node is in (lines 16– 19 in the Algorithm 4.7).

4.3.4 Combining heuristics

The different heuristics can be combined to achieve different outcomes in specific environments. These combinations were evaluated, and the results are presented in chapter 5 for generic and high-performing

Algorithm 4.7: Build an informed tree

```
Input: A list $B_i$ with the group of $I$ internal nodes. $C$ - number of $\delta$-aglomerates. $bk$ - block size.
Output: Tree $T$.
/* Add root to tree                                                    */
1  $root \leftarrow rootHeuristic()$;
2  $T \leftarrow \{root\}$ ;
/* Global dissemination on the first level                             */
3  for $internal = 1, 2, \ldots C$ do
4      $root.children \leftarrow root.children \cup$ {a node from $\delta$-aglomerate $C_i$ not chosen from group $B_i$}
/* Calculate wanted fanout per $\delta$-aglomerate                     */
5  for $i \in$ range($C$) do
6      $M[i] \leftarrow \frac{bandwidth(i)}{bk}$;
/* Local dissemination on the first level                              */
7  $available \leftarrow$ nodes from group $B_i$ from the same location of $root$;
8  while $available! = \emptyset$ do
9      for $\ldots M[root.location]$ do
10         $root.children \leftarrow root.children \cup$ { node from $available$ not chosen };
/* Create the remaining subtrees                                       */
11 for $internal = 1, 2, \ldots root.children$ do
12     if $internal.location! = root.location$ then
13         $available \leftarrow$ nodes from group $B_i$ from the same location of $internal.location$;
14         while $available! = \emptyset$ do
15             for $\ldots M[internal.location]$ do
16                 $internal.children \leftarrow internal.children \cup$ { node from $available$ not chosen };
/* Add leafs                                                           */
17 for $deepestInternal \in T$ do
18     for $\ldots M[deepestInternal.location]$ do
19         $deepestInternal.children \leftarrow deepestInternal.children \cup$ { node from remaining nodes
           not chosen };
20 return $T$
```

deployments.

Summary

In this chapter, we presented the architecture of Kauri-Adaptive composed of heuristics to build topology-aware trees that leverage the heterogeneity of the environment. In the next chapter, we present the experimental evaluation of these heuristics.

5 Evaluation

Contents

5.1 Experimental goals

In our evaluation, we wish to answer several questions, specifically about the gains and disadvantages of our implemented heuristics compared to the state of the art. More precisely, we address the following questions:

- What is the impact of our heuristics in heterogeneous deployments?
- Is the sequence of leaders known to an attacker?
- Is the system fair?

To answer the last two, we look at the sequence phase described in Section 4.2.2 and guarantee that by using the VRF, the sequence will not be deterministic, and an attacker will not be able to exploit. Also, if the function used in VRF follows a uniform distribution, we can say that we achieve fairness in the long term.

Next, we will analyze the impact of the heuristics used in the generation phase.

5.2 Experimental workbench

The experimental evaluation was conducted using a discrete event simulator in Python already used in other works [2, 3]. The simulated network model follows the latency and bandwidth matrix described in Table 4.1, with the processes distributed evenly across the six data centres. To add the bandwidth model to the existing simulator, we implemented an algorithm that models the flow of messages in a simulation, accounting for bandwidth and latency, to simulate the order and timing of message delivery between nodes. Each node in the simulation maintains sender and receiver channels. When a message is sent from a source node to a destination node, the algorithm calculates transmission times based on message size and bandwidth between them. It schedules message transmission and reception events, considering current time and channel availability to ensure proper sequencing. Messages are processed upon delivery.

The implemented approach is described in detail in Algorithm 15.

5.3 Preliminary Experiments

To validate the use of this previously described simulator, we conducted some preliminary experiences to compare its results with the published results from Kauri [4]. From the scenario mentioned as ”Heterogeneous Networks” in Kauri, we set up a similar test with $N = 60$ participants distributed through the

Algorithm 5.1: Bandwidth logic added to the simulator used in other work [2, 3].

```
Input: Each message $m$ contains $m.src$ - source node, $m.dest$ - destination node, $m.sz$ -
       message size. $bandwidth(s,d)$ - bandwidth between s and d, $latency(s,d)$ - latency
       between s and d
for each node $n$ do
    $n.senderChannelFree = 0$;
    $n.receiverChannelFree = 0$;
if send (m) then
    $transmissionTime = m.sz/bandwidth(m.src, m.dest)$;
    $startSend = max(NOW, m.src.senderChannelFree)$;
    $m.src.senderChannelFree = startSend + transmissionTime$;
    trigger $firstBitReady(m)@time = startSend + latency(m.src, m.dest)$;
if firstBitReady(m) then
    $transmissionTime = m.sz/bandwidth(m.src, m.dest)$;
    $startReceiving = max(NOW, m.dest.receiverChannelFree)$;
    $m.dest.receiverChannelFree = startReceiving + transmissionTime$;
    trigger $deliver(m)@time = m.dest.receiverChannelFree$;
if deliver (m) then
    process the message $m$;
```

datacenters from Table 4.1. We simulate a star topology representing Hotstuff [6] and a tree representing Kauri [4]. The leader is a node in Oregon for both cases.

We evaluated that we reached similar results as the ones published in Kauri. Regarding latency, HotStuff outperforms Kauri because the system size is small, so bandwidth does not become a bottleneck. However, HotStuff's latency would quickly grow with the system as bandwidth becomes the bottleneck, while Kauri's latency would remain largely unaffected. Nonetheless, we call out that Kauri only creates a $\approx$ 2x increase in latency with this simulator, leading us to conclude that it is a good instrument to use in the following scenarios.

5.4 Generic Deployment

We coded the algorithms described in Section 4.3 in the simulator. We considered three typical scenarios: a scenario in which the fanout defined by the administrator coincides with the number of δ-agglomerates, a scenario in which the fanout is higher and another in which it is lower. As the primary metric, we used the time needed for the root to collect the quorum, which is the instant from which the consensus algorithm can make progress.

To independently assess the impact of the heuristic for allocating nodes to groups and building trees from a group, we evaluated the following combinations for each scenario: i) informed groups with informed trees; ii) informed groups with random trees; iii) random groups with informed trees; iv) random groups with random trees.

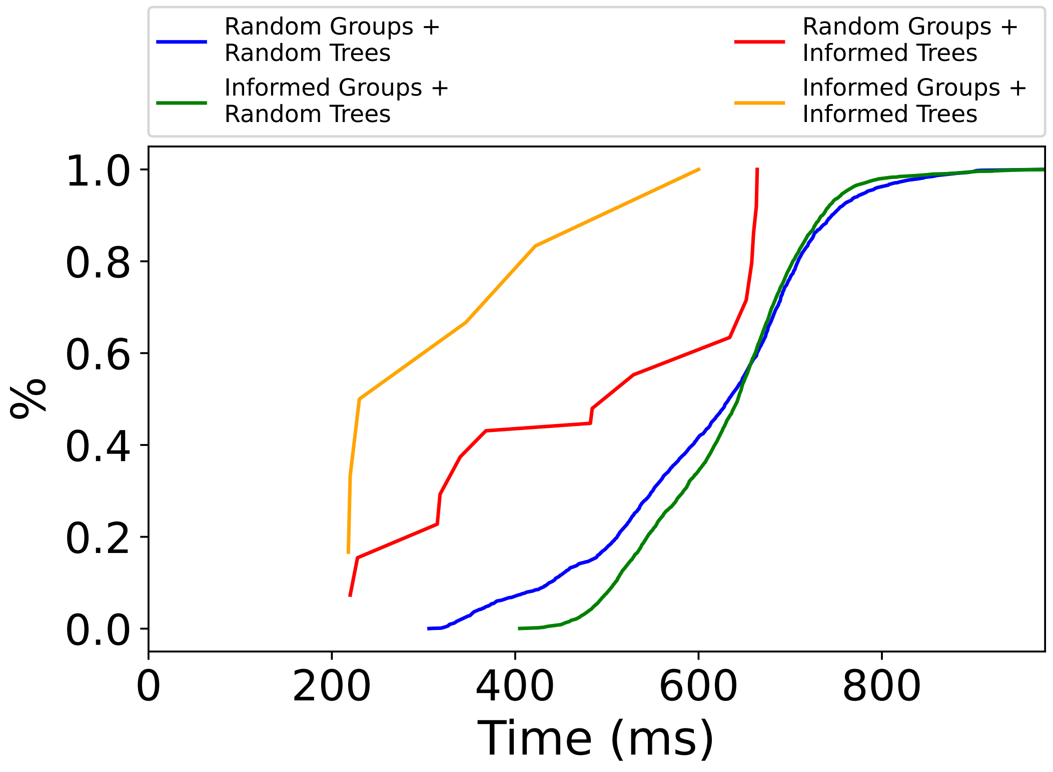


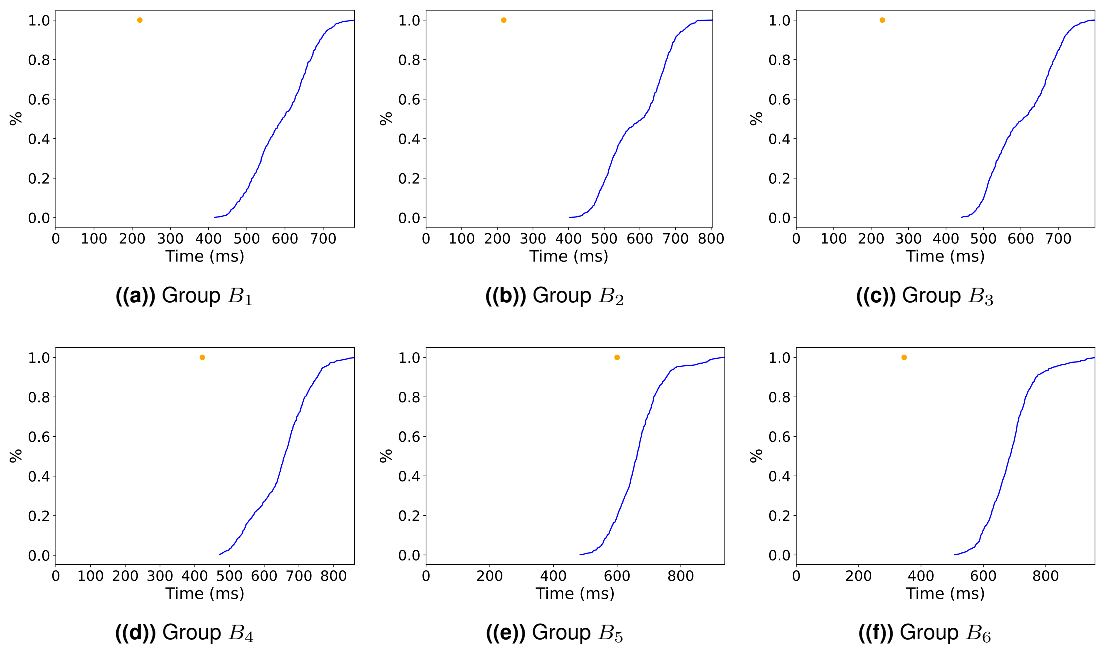
Figure 5.1: Time to retrieve quorum in Scenario 1.

5.4.1 Scenario 1: $M =$ number of δ-agglomerates

With 6 δ-agglomerates for this scenario, we set the fanout $M = 6$ and create a regular two-level tree. For this purpose, we have $N = 43$ nodes. We generated ten groups for the random groups, and for the random trees, we generated 100 trees for each group. The results for this scenario in the four combinations are shown in Figure 5.1.

Starting with constructing random trees (green and blue lines), we can see that they perform worse than informed solutions. The differences between building random trees from a random or informed group distribution are not significant, and, in the case of this figure, the difference between the results is because we could not take a large enough sample. As we see below, these lines appear increasingly overlapping in the other scenarios. On the other hand, the informed algorithm for building trees already shows significant gains, even when the constitution of the groups is still random; in this case, we see an increase in performance by 30% in half of the trees. The informed distribution of the nodes among the groups improves the results even more (in the order of 70%), as it increases the probability of a well-situated root in all the groups.

Figure 5.2 shows the impact of combining the different heuristics. In particular, we selected a distribution of nodes by groups and compared the informed tree built by our heuristic from each group (yellow



((a)) Group B_1 **((b))** Group B_2 **((c))** Group B_3

((d)) Group B_4 **((e))** Group B_5 **((f))** Group B_6

Figure 5.2: Informed heuristic (yellow) vs random strategy (blue) in Scenario 1.

dot) with several random trees that can be made from the same group (blue line). As we can see, for most groups, the informed tree performs better than any of the 100 randomly constructed trees, which shows that the probability of obtaining a good tree in an uninformed way is generally low. However, the figure also shows that our heuristic does not always get the best tree. For example, in Figure 5.2(e), group B_5 includes two nodes in Taiwan and one node in each of the other δ-agglomerates. Although the δ-agglomerate with the lowest average latency for the additional δ-agglomerates in this scenario is in Iowa, finding a more efficient tree with the root in Taiwan is possible.

5.4.2 Scenario 2: $M >$ number of δ-agglomerates

Keeping the 6 δ-agglomerates used previously, we set the fanout $M = 10$ and created a two-level regular tree. For this purpose, we used $N = 111$ nodes. The results for this scenario are shown in Figures 5.3 (aggregated results for various distributions of nodes per group) and Figure 5.4 (for a particular distribution). As we can see, the various solutions show the same relative pattern, although in this case, the advantages of informed trees are more evident. This is because, by having a larger fanout, we have a greater parallelism factor in the dissemination, which is better exploited by the informed trees.

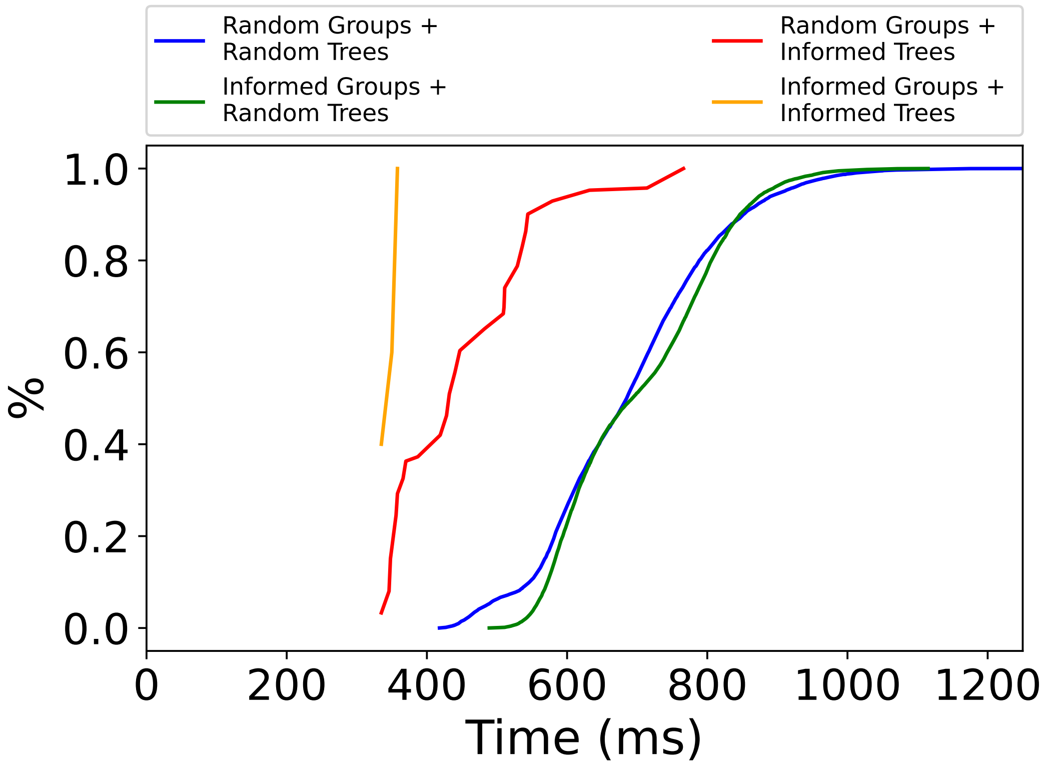
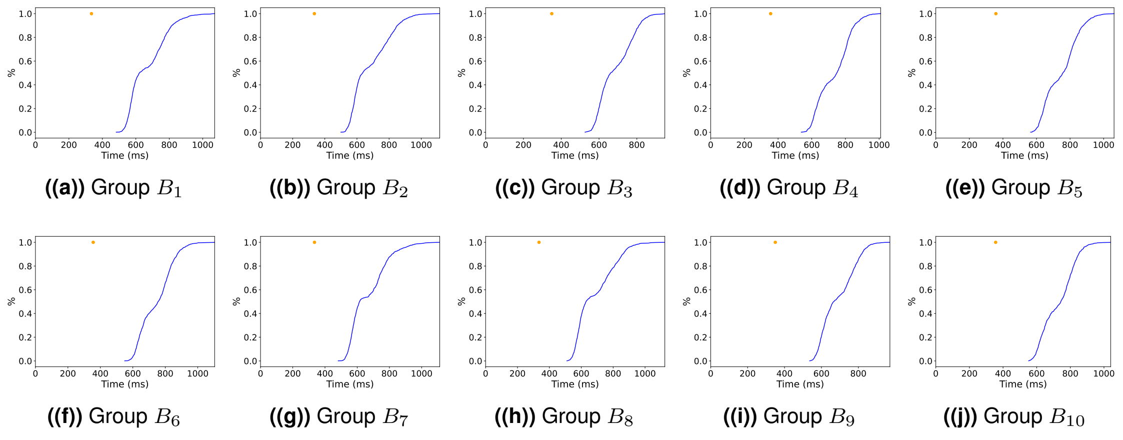


Figure 5.3: Time to retrieve quorum in Scenario 2.



((a)) Group B_1 ((b)) Group B_2 ((c)) Group B_3 ((d)) Group B_4 ((e)) Group B_5 ((f)) Group B_6 ((g)) Group B_7 ((h)) Group B_8 ((i)) Group B_9 ((j)) Group B_{10}

Figure 5.4: Informed heuristic (yellow) vs random strategy (blue) in Scenario 2.

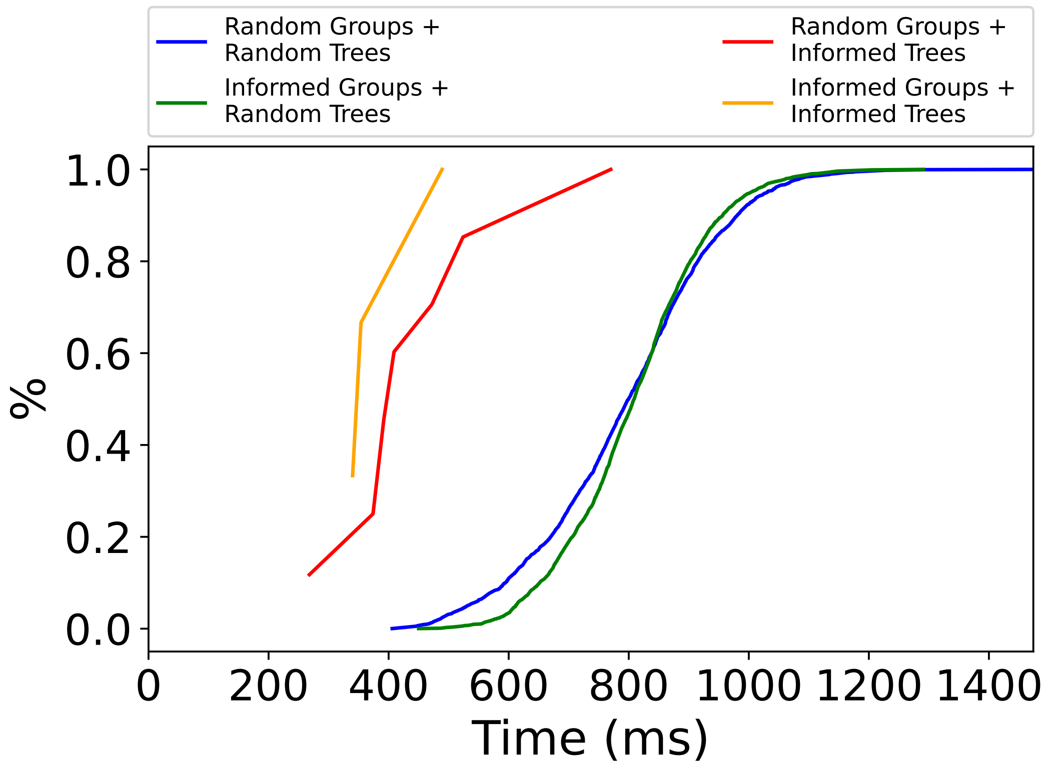
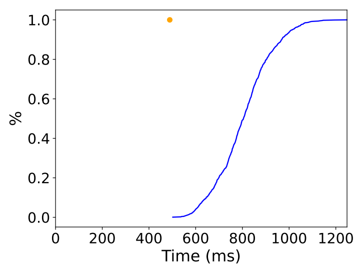


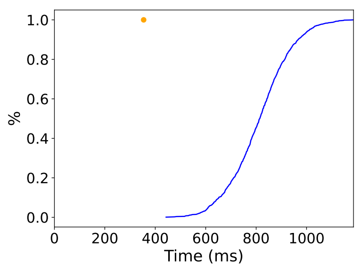
Figure 5.5: Time to retrieve quorum in Scenario 3.

5.4.3 Scenario 3: $M <$ number of δ-agglomerates

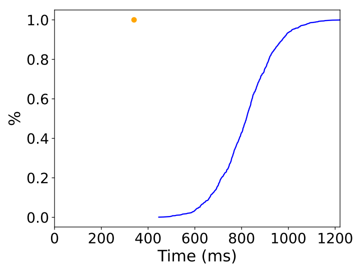
Finally, we evaluate a scenario in which the fanout is smaller than the number of δ-agglomerates, which typically implies increasing the depth of the tree. To do this, we keep the 6 δ-agglomerates, set the fanout $M = 3$ and create a regular 3-level tree. For this purpose, we used $N = 40$ nodes. The results for this scenario are shown in Figure 5.5 and Figure 5.6, using the same methodology as in the previous scenarios. Once again, it is clear that using informed strategies allows for shorter collection times. However, in this scenario, the difference between using an informed or random distribution of the nodes among the groups is insignificant. This is partly explained by the same reasons as in the previous scenario - by having a smaller fanout, informed trees can exploit the parallelism factor less, which dilutes the difference between informed and random groups. It should be noted, however, that increasing the depth also impacts the results that require further study.



((a)) Group B_1



((b)) Group B_2



((c)) Group B_3

Figure 5.6: Informed heuristic (yellow) vs random strategy (blue) in Scenario 3

5.5 High-performing Deployment

We coded the algorithms described in Section 4.3 regarding the heuristic using the local-dissemination approach. The same simulator, metrics and informed group distribution were used with the nodes uniformly distributed between the 6 data centres. For 430 nodes, messages of size 2000 Mbit/s and the Bandwidth-aware Distribution of nodes in groups described in Section 4.3. The proposed solution, where we fully leverage the local dissemination and create subtrees with fanout based on the bandwidth bottleneck of each cluster from a root from the cluster with the most bandwidth, was compared with a random distribution approach where we create 1000 random trees as follows: we connect the root to the internal nodes in the group and for the leaves, we randomly distribute them by a fixed fanout.

We gathered the results for a group in Figure-5.7. As expected from the previous heuristic results, the informed heuristic outperforms the random approach. For half the random trees, the informed heuristic can outperform the other by 80%. Even though the testbed of 1000 random trees could be further improved, it seems that in this heuristic, it will be much harder to find cases like in Figure 5.2(e)) where we have random trees with better performance than those found by our heuristic.

5.6 Discussion

For the generic deployment, the evaluation considers a relatively small set of scenarios. Still, it covers different relationships between the number of δ-agglomerates and the fanout of the tree, i.e. scenarios in which the fanout is equal (Scenario 1), higher (Scenario 2) or lower (Scenario 3) than the number of δ-agglomerates. In all scenarios, our generic tree-building heuristic achieves, on average, much shorter collection times than the random trees currently used by Kauri. Furthermore, in all cases, the informed division of nodes into groups improves the results of the heuristic used to build a tree in an informed way. This effect is less pronounced in Scenario 3. Still, more experiments would be needed to understand whether it is the value of the fanout or the depth of the tree that most affects the impact

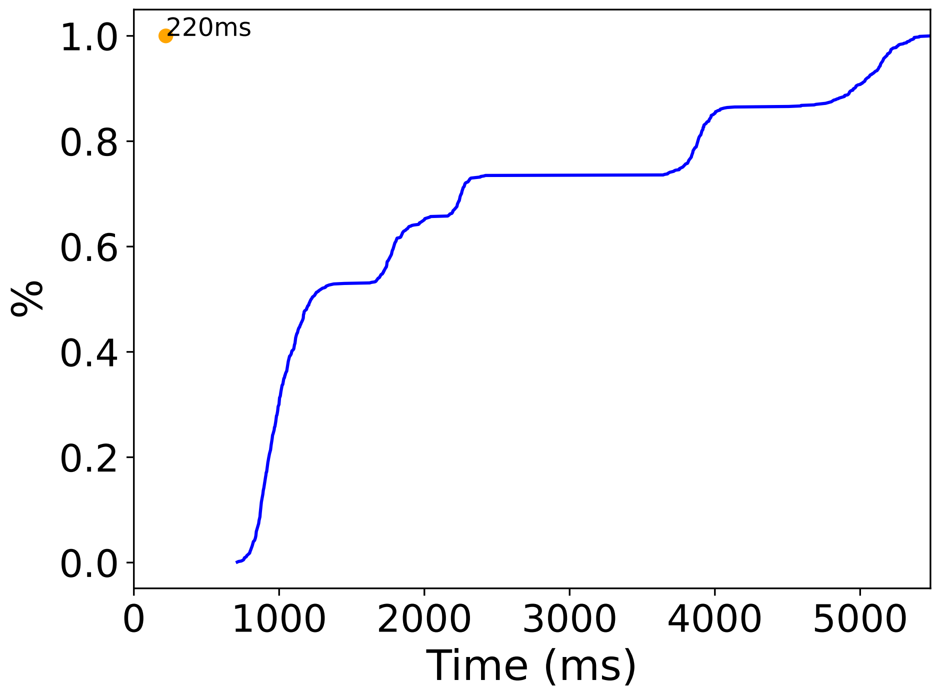


Figure 5.7: Informed heuristic (yellow) vs random strategy (blue).

of the informed distribution of nodes by groups. The evaluation also shows clusters for which it is easy to find, even at random, trees with better performance than those found by our heuristic (for example, the case presented in Figure 5.2(e)). We are investigating whether it is possible to improve the generic heuristic to avoid these cases without making it unnecessarily complicated (an advantage of the current version is that it is very efficient to run from a computational point of view).

For the last deployment, we still registered a significant improvement in collection times than random trees. Even though we have to expand the experiments to understand the impact with different bandwidth models and message sizes, the scenario indicates we can find a tree better than the random best case in a determinist and computationally efficient manner.

Summary

This chapter presents our evaluation, starting with our goals and experimental workbench. We showed results for a generic deployment and a high-performing deployment. In all scenarios, our tree-building heuristic achieves, on average, much shorter collection times than the random trees currently used by Kauri. In the next chapter, we present our conclusions, current system limitations, and future work for overcoming these limitations.

6

Conclusion

Most permissioned blockchains are based on variants of BFT consensus protocols that have scalability problems since all participants engage in multiple rounds of data exchange. Kauri [4] combines dissemination/aggregation trees and pipelining to distribute the load and compensate for the increased latency of using trees. However, it uses a stable-leader approach to leverage the benefits of pipelining. Several arguments favor changing the leader between consecutive consensus rounds to provide censorship resistance. These arguments motivate the need for a strategy to rotate the leader with a minimal negative impact on pipelining and promoting a good load balance among all participants.

We surveyed the state-of-the-art BFT algorithms. We analyzed their communication patterns and discussed leader rotation strategies, the impact of the pipelining techniques and how they behave in heterogeneous deployments.

In this work, we present a solution that aims to enrich Kauri with heuristics for generating dissemination and aggregation trees based on the network topology. In this context, we propose a heuristic for building trees that uses information about the latency between nodes to create groups and another for generating trees from these groups. We evaluate both heuristics using simulations based on realistic network latencies and present results that suggest it is possible to reduce the time needed to collect a Byzantine quorum by 70%. We elaborated a solution to support a rotating-leader approach in a tree topology that leverages these heuristics while maintaining an acceptable throughput by the pipelining technique.

In future work, we intend to evaluate our approach in a real code scenario running in geo-distributed data centres. Another relevant scenario would leverage pipelining techniques to assess how this rotating strategy behaves. In the generic heuristic, we also consider that the system administrator provides the M and δ parameters; it would be interesting to extend the work to configure these parameters automatically. As mentioned in the evaluation chapter 5, the impact of increasing the depth of the trees when leveraging these heuristics needs to be further studied to understand if it is the fanout of the tree or its depth that has the most impact. Also, for the clusters where it is easy to find a random tree that outperforms one using our generic heuristic, we need to investigate whether it is possible to improve our generic heuristic to avoid those cases without incurring a much higher computational cost. Lastly, in our last heuristic, we must expand the testbed to include different bandwidth models and message sizes in real-world scenarios.

Bibliography

[1] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, “Resilientdb: Global scale resilient blockchain fabric,” *arXiv preprint arXiv:2002.00160*, 2020.

[2] M. Matos, P. Felber, R. Oliveira, J. O. Pereira, and E. Rivière, “Scaling up publish/subscribe overlays using interest correlation for link sharing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2462–2471, 2013.

[3] M. Matos, H. Mercier, P. Felber, R. Oliveira, and J. Pereira, “Epto: An epidemic total order algorithm for large-scale distributed systems,” in *Proceedings of the 16th Annual Middleware Conference*, ser. Middleware ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 100–111. [Online]. Available: https://doi.org/10.1145/2814576.2814804

[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*, 2021, pp. 35–48.

[5] M. Castro, B. Liskov *et al.*, “Practical byzantine fault tolerance,” in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[6] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, “Hotstuff: Bft consensus in the lens of blockchain,” *arXiv preprint arXiv:1803.05069*, 2018.

[7] H. Teixeira, L. Rodrigues, and M. Matos, “Arvores de disseminaçao e agregaçao cientes da topologia para suportar consenso bizantino em larga escala,” *Inforum*, 2023.

[8] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” *Decentralized Business Review*, p. 21260, 2008.

[9] 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.

[10] D. Yaga, P. Mell, N. Roby, and K. Scarfone, “Blockchain technology overview,” *arXiv preprint arXiv:1906.11078*, 2019.

[11] É. R. Keresztes, I. Kovács, A. Horváth, and K. Zimányi, “Exploratory analysis of blockchain platforms in supply chain management,” *Economies*, vol. 10, no. 9, p. 206, 2022.

[12] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[13] E. Kokoris-Kogias, “Robust and scalable consensus for sharded distributed ledgers,” *Cryptology ePrint Archive*, 2019.

[14] 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.

[15] G. Zhang and H.-A. Jacobsen, “Prosecutor: An efficient bft consensus algorithm with behavior-aware penalization against byzantine attacks,” in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 52–63.

[16] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, “Spin one’s wheels? byzantine fault tolerance with a spinning primary,” in *2009 28th IEEE International Symposium on Reliable Distributed Systems*. IEEE, 2009, pp. 135–144.

[17] Y. Amir, B. Coan, J. Kirsch, and J. Lane, “Byzantine replication under attack,” in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 197–206.

[18] T. H. Chan, R. Pass, and E. Shi, “Pala: A simple partially synchronous blockchain,” *Cryptology ePrint Archive*, 2018.

[19] I. Abraham, N. Crooks, N. Giridharan, H. Howard, and F. Suri-Payer, “It’s not easy to relax: liveness in chained bft protocols,” *arXiv preprint arXiv:2205.11652*, 2022.

[20] S. Alqahtani and M. Demirbas, “Bottlenecks in blockchain consensus protocols,” in *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*. IEEE, 2021, pp. 1–8.

[21] S. Cohen, R. Gelashvili, L. K. Kogias, Z. Li, D. Malkhi, A. Sonnino, and A. Spiegelman, “Be aware of your leaders,” in *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*. Springer, 2022, pp. 279–295.

[22] P. Li, G. Wang, X. Chen, F. Long, and W. Xu, “Gosig: a scalable and high-performance byzantine consensus for consortium blockchains,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 223–237.

[23] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, “Algorand: Scaling byzantine agreements for cryptocurrencies,” in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 51–68.

[24] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzzyva: speculative byzantine fault tolerance,” in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 45–58.

[25] X. Sui, S. Duan, and H. Zhang, “Marlin: Two-phase bft with linearity,” *Cryptology ePrint Archive*, 2022.

[26] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, “Making byzantine fault tolerant systems tolerate byzantine faults.” in *NSDI*, vol. 9, 2009, pp. 153–168.

[27] 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)*, 2016, pp. 279–296.

[28] J. Seibert, S. Becker, C. Nita-Rotaru, and R. State, “Newton: Securing virtual coordinates by enforcing physical laws,” *IEEE/ACM Transactions on Networking*, vol. 22, no. 3, pp. 798–811, 2014.

[29] R. Albert, H. Jeong, and A.-L. Barabási, “Diameter of the world-wide web,” *nature*, vol. 401, no. 6749, pp. 130–131, 1999.

[30] S. E. Schaeffer, “Graph clustering,” *Computer science review*, vol. 1, no. 1, pp. 27–64, 2007.