Dynamic Trees for Byzantine Consensus Protocols

PIC2 - Master in Computer Science and Engineering
Instituto Superior Técnico, Universidade de Lisboa

Tomás Pereira - 95682 [*]
`tomas.araujo.pereira@tecnico.ulisboa.pt`

INESC-ID, Instituto Superior Técnico, U.Lisboa
Advisors: Prof. Luís Rodrigues and Prof. Miguel Matos

Abstract

This work addresses the reconfiguration of leader-based Byzantine Fault Tolerant (BFT) consensus protocols that use dissemination/aggregation trees to support the communication between the leader and the remaining participants. We consider a stable-leader approach, where a given configuration is used for running multiple consecutive instances of consensus until it is suspected to be faulty. The goal is to design a reconfiguration strategy that promotes the use of efficient topologically aware trees. Furthermore, if the nodes suspected to be faulty are inner nodes in the tree, we aim to perform reconfiguration at the local level on the sub-branches of the tree, to maximize the number of nodes that are not affected by the reconfiguration.

Keywords — Distributed Systems, Blockchain, Byzantine Fault Tolerance, Consensus

[*]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 (`https://nape.tecnico.ulisboa.pt/en/apoio-ao-estudante/documentos-importantes/regulamentos-da-universidade-de-lisboa/`).

Contents

1 Introduction

Byzantine fault tolerant (BFT) consensus protocols allow correct processes to reach agreement even in the presence of a fraction of malicious processes. BFT protocols have been first introduced for synchronous systems [1] but have been subsequently extended to execute in eventually synchronous settings [2, 3, 4]. BFT protocols require multiple rounds of message exchange among participants and are, therefore, costly both in terms of communication and processing. For this reason, early implementations considered a relatively small number of participants (in the order of dozens) [5]. However, with the emergence of the blockchain abstraction and the increasing relevance of large-scale systems based on blockchains, the need to design byzantine consensus protocols that can scale to hundreds of participants has been an emerging topic.

PBFT [2], one of the first BFT consensus protocols designed for eventually synchronous systems, is a leader-based protocol that uses an all-to-all communication pattern, i.e., there are coordination rounds where participants send (and receive) messages to (from) every other participant. Due to the use of this communication pattern, PBFT is inherently non-scalable. Several approaches have emerged to circumvent the scalability limitations of PBFT-like protocols. One approach consists of using a star-based communication pattern, such as HotStuff [3], where nodes only communicate directly with the leader: this reduces the message complexity from quadratic to linear, but the leader remains a bottleneck. Another approach is to use hierarchical strategies, such as in Fireplug [6], where consensus is achieved by the hierarchical combination of several sub-consensus instances that are executed among smaller sub-groups. A limitation to approaches of this nature is how they reduce the resilience of the system, given that the fraction of byzantine nodes required to take control of a sub-group is smaller than the fraction of nodes required to prevent consensus in the super-group. Once a sub-group is compromised, consequently the consensus in the super-group is as well. Finally, the use of dissemination and aggregation trees has been proposed to circumvent the scalability limitations of the star-based communication pattern, while retaining the resilience properties of non-hierarchical approaches. In our work, we will focus on protocols that use this strategy.

Kauri [4] is a BFT consensus protocol that extensively uses dissemination and aggregation trees to achieve scalability. Since the use of trees introduces extra latency in the communication, Kauri also uses an aggressive pipelining strategy that expands on the pipelining already used in protocols such as HotStuff, allowing the leader to start multiple consensus instances before previous instances have terminated. Kauri has two main limitations. First, it is designed for homogeneous settings and its trees are constructed using randomization, which does not take into account the capacity of the nodes or the latency present in the communication links. Secondly, when a reconfiguration is required, a tree is replaced by a completely different tree that typically does not share any inner node with the previous tree. While this reconfiguration strategy permits Kauri to find a robust tree quickly, it is highly disruptive for the pipelining process and hence for performance.

In this work, we address the two main limitations of Kauri identified above. On one hand, we aim to design algorithms that optimise dissemination/aggregation trees by leveraging information about the network latencies and the CPU capacity of the nodes. On the other hand, we aim to design reconfiguration strategies that can be applied to sub-branches of the tree, to maximize the number of nodes that are not affected by the reconfiguration.

The rest of the document is structured as follows: Section 2 summarizes the goals and the expected results of this project. In Section 3, we provide more insight in regard to relevant blockchain BFT consensus concepts. In Section 4, we survey relevant BFT consensus protocols with an emphasis on efficiency and performance. In Section 5, we aim to define a high-level and preliminary approach to solve our reconfiguration problem. In Section 6, we define our evaluation metrics, methodology, and goals. In Section 7, we present the schedule of future work. And lastly, in Section 8, we provide a conclusion to this report.

2 Goals

Our work addresses the construction and reconfiguration of dissemination/aggregation trees to support scalable byzantine fault-tolerant consensus. More precisely:

> *Goals:* We aim at using information regarding the network latency between consensus participants and information regarding their performance to build dissemination/aggregation trees that support the fast exchange of information among a quorum of the participants. Furthermore, if one or more inner nodes of the tree are suspected to be faulty or showing poor performance, we aim at designing reconfiguration strategies that can mitigate the problem locally and avoid a global reconfiguration unless the leader is suspected to be faulty.

Consequently, the project is expected to produce the following results:

> *Expected Results:* The work will produce i) a specification of a tree construction algorithm and of its reconfiguration algorithm; ii) an implementation of the proposed algorithms, iii) an experimental evaluation of the performance of the resulting implementation.

3 Background

The objective of this section is to introduce the required background for our work. Section 3.1 introduces the blockchain abstraction, Section 3.2 defines Byzantine Fault Tolerant Consensus and Section 3.3 discusses the operation of consensus protocols in wide-area network (WAN) deployments.

3.1 Blockchain Systems

The term *blockchain* refers to an abstraction that offers a persistent, append-only, totally ordered ledger of records. Records are grouped into *blocks*, that can be added to the head of the ledger. Once a new block is *committed*, it can no longer be deleted or tampered with. Thus, the sequence of committed blocks is immutable, and the ledger can only be updated by adding additional blocks to the ledger itself. This ledger is typically implemented as a linked-list of blocks, where each block has a reference to their predecessor block in the ledger, thus creating a chain of blocks. Although it is possible to materialize this abstraction using a centralized server, blockchain implementations are generally distributed and decentralized. In this case, the ledger is cooperatively maintained by a possibly large and open set of nodes that coordinate to commit new blocks to the chain and to prevent committed blocks from being tampered with. Using the appropriate protocol, it is possible to derive a reliable and trusted decentralized implementation of the blockchain in settings where a fraction of the participants may not be trusted.

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

Permissionless blockchains: In a permissionless blockchain, any node in the system can participate in the coordination protocol used to maintain the chain. Decentralized ledgers were originally popularized with a permissionless setting, as used in Bitcoin [7]. In this setting, a node can join or leave the protocol at any moment, and the number of nodes participating in the protocol is not limited. Additionally, individual nodes cannot be trusted but it is assumed that a majority of nodes are correct and do not depict malicious faults. Malicious nodes can aim to overthrow the system, by discarding blocks proposed by correct nodes and by attempting to prevent correct nodes from proposing new blocks. For this purpose, malicious nodes may also adopt multiple identities to appear as multiple nodes, a behaviour that is often referred to as a Sybil attack [8]. To limit the power of malicious nodes, and in particular, to limit their ability to propose new blocks, a participant must commit a substantial number of resources to the protocol using techniques such as Proof-of-Work (PoW) [7] or Proof-of-Space [9]. Unfortunately, the use of techniques such as PoW limits the throughput of the system and induces substantial energy consumption, making these protocols ecologically unfriendly. Another limitation of techniques such as

PoW is that parties may benefit if they collude to solve the algorithm's crypto-puzzle, which creates an incentive to reduce decentralization.

Permissioned blockchains: Permissioned blockchains have gained traction due to their potential to create more efficient and secure systems when compared to permissionless designs [10]. In a permissioned blockchain, participants are known and limited, creating a scenario where the system has controlled decentralization but that also performs better, as mechanisms for Sybil attack resistance are no longer required. Additionally, permissioned blockchains allow for the implementation of BFT consensus protocols, which, combined with the fact that participants have to be authorized, creates a much more controlled environment for system execution. BFT consensus alternatives found in permissioned blockchains are also known to be much more energy efficient when compared to protocols found in permissionless settings, due to the fact that in the latter security is often assured through the use of computationally heavy techniques and algorithms.

3.2 Byzantine Fault Tolerant Consensus

As mentioned above, BFT consensus protocols are a popular solution in permissioned blockchain contexts. The goal of BFT consensus is to allow correct nodes to agree on a singular valid output for an instance of consensus, under the assumption that there may be f nodes exhibiting arbitrary behaviour. While arbitrary behaviour may refer to any type of failure where any error may occur, in the Byzantine Fault Tolerant model we assume the worst-case scenario where faulty nodes may collude with malicious intent to overthrow the system [11].

In a BFT system with N nodes, we say that we have $N = 3f + 1$ resilience when the system is able to reach consensus in the presence of $f = \lfloor \frac{N-1}{3} \rfloor$ faulty nodes. This is done by leveraging *quorums* of size $Q = 2f + 1$ for coordination, which guarantees that at least one correct node belongs to any pair of quorums, enabling them to relay protocol messages from one quorum to another. Byzantine Fault Tolerant Consensus is characterized by the following properties [11]:

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

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

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

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

Definition 2.1 (Weak Validity). *If all processes are correct and propose the same value v, then no correct process decides a value different from v; furthermore, if all processes are correct and some process decides v, then v was proposed by some process.*

Definition 2.2 (Strong Validity). *If all correct processes propose the same value v, then no correct process decides a value different from v; otherwise, a correct process may only decide a value that was proposed by some correct process or the special value $\square$.*

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

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

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

Integrity and Agreement, alongside a Validity variant, are the properties that uphold safety throughout consensus execution. Adoption of such properties makes it so consensus instances have a deterministic effect on system state when combined with the use of State Machine Replication (SMR) techniques to propagate the agreed-upon changes and commands to all the system's participants. Furthermore, the combination of these properties makes it so that the agreed-upon state has been witnessed and approved by all correct processes.

3.3 WAN Deployment

Permissioned blockchains are frequently deployed within wide-area networks (WANs). As the name implies, WANs are a type of network that connects different devices over a large geographic area, sometimes even globally. WANs may reveal themselves to be unreliable and create difficult conditions to both guarantee system liveness and achieve good performance, due to the common occurrence of network partitions and due to the usually high latency and low bandwidth that systems must face. These traits accentuate the fact that in most real-world scenarios, network links in a WAN are inherently asymmetric and create a heterogeneous topology, which consequently increases the asynchrony between pairs of nodes. When deployed in these conditions, it is standard for systems to take into account mechanisms that can compensate for and mitigate the downsides of the geographic distance between devices. Blockchains are no exception to this, existing several works that tackle the scalability and reliability of consensus in this setting, such as Steward [12], Mencius [13], and the recently proposed notion of Planetary Scale Byzantine Consensus proposed by Voron *et al.* [14].

4 Related Work

Given the background provided by Section 3, we can now discuss the relevant features frequently utilized in the design and implementation of state-of-the-art permissioned blockchain BFT protocols. We will first enumerate important traits that help distinguish protocols apart and then look into several examples of BFT protocols that leverage these traits to improve the efficiency and reliability of protocol execution, namely scalability, reduced communication complexity, reduced latency, and increased system throughput. Finally, we will provide an overarching discussion that will help highlight which beneficial features our solution aims to achieve.

4.1 Towards Coordinated Agreement in Blockchain

With the formal definition for BFT consensus defined in Section 3.2, it is important to differentiate and detail some of the more common approaches to achieve agreement between nodes in a distributed system. It is of note that, in a blockchain environment, consensus is run to agree on which transactions will be appended to the distributed ledger and in which order they will be committed and executed. This means that at the heart of consensus applied to blockchain, protocol designs need to take into account mechanisms that ensure coordination between correct nodes for consecutive consensus instances. Through proper coordination, distributed consensus systems are able to commit and execute client requests, avoid conflicts and achieve agreement with reduced message and time complexities [15]. We define the following approaches for coordinated agreement:

Leader-based [2, 3, 4]. In leader-based BFT consensus algorithms, the role of the coordinator, or as it is usually called, the leader, is to help all correct processes converge towards a decision in a relatively fast manner [16]. In the context of blockchain, it is the leader's role to propose which transactions will be appended to the ledger. Consequently, this means that system progress is directly linked with leader performance. Leaders in consensus are usually defined through the use of *view synchrony* algorithms [11], where the transition from one view to the next dictates a leader change. Due to the importance of the leader role, systems must provide mechanisms to define the leader election policy and to efficiently recover from consensus executions where the leader is found to be faulty. Additionally, in the context of blockchains, the existence of a system-wide leader creates concerns regarding censorship and leader fairness, which has led to the creation of more sophisticated algorithms that aim to mitigate these issues [17, 18]. Leader-based algorithms typically fall under two categories: *efficient leader-based*, where the leader's role is correlated with achieving a target performance with high throughput and low latency in fault-free executions; and *robust leader-based*, where the leader's role is providing high fault-tolerance to the system instead of achieving maximum performance [15].

Leaderless [19, 20]. Leaderless coordination aims to mitigate the previously mentioned issues regarding leader-based systems, such as censorship and leader-related bottlenecks, thus obtaining higher decentralization [16, 15]. Additionally, by providing equity in regard to the task of transaction proposal, these systems often aim at utilizing this parallelization to obtain gains in the system's transaction throughput. The main disadvantage of leaderless coordination ends up being the complexity of the mechanisms required to reach proper agreement and ordering, especially in the context of blockchains, where blocks need to know the previous block's hash. The lack of a primary node acting as a coordinator in the system creates conflicts related to proposal order. Consequently, solutions end up having increased message and time complexities due to the mechanisms which are usually applied to either avoid or correct such ordering conflicts.

Multi-leader [21, 22]. Multi-leader coordination can be defined as a specification of leaderless design, where instead of giving the ability to propose simultaneously to all system participants, there is a defined set of leaders assigned to a portion of system execution. The multi-leader coordination approach aims at bridging both the advantages of leader-based and leaderless protocols: it utilizes the additional concurrency to achieve a better throughput and avoid bottlenecks while simultaneously limiting the amount of leaders in the system to reduce the higher communication overheads and conflict rates seen in leaderless designs. However, unlike leaderless systems, multi-leader protocols must take into account election mechanisms and leader selection policies, which come with additional challenges in terms of algorithm complexity [23].

Group-based [24, 6]. In group-based coordination, the system is divided into different groups that align with either different responsibilities or different node characteristics, such as geographic location or latency. Afterwards, these groups may be used to allow correct nodes to reach agreement through the use of known coordination mechanisms. A common design example of group-based coordination is the definition of a *global* group whose nodes are delegated with the management of a dedicated *local* group. Each of the global group's nodes may perform tasks such as the collection and dissemination of their local group's votes. Consensus is reached through the global group's communication channels and the results are later propagated to the lower hierarchy local groups. Although group-based systems can offer better scalability and performance, these qualities can only be achieved under efficient grouping assumptions, which is a highly complex task when taken into account in a real-world context [23].

4.2 Communication in Blockchain Consensus

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

4.2.1 Timing Assumptions

As expected of most real-world scenarios, distributed systems must be defined within the parameters of real-world network systems, and blockchains are no exceptions. BFT consensus systems usually make one of the three following assumptions regarding the time bounds on communication delays [11]:

Asynchronous Systems. The basis for asynchronous systems can be summarized as not making any timing assumptions about processes and links. The passage of time is usually based on the transmission and delivery of messages, opposite to what happens in synchronous systems where participants can accurately time their assumptions by relying on physical clocks. It is still possible to capture some notion of the passage of time by encoding cause-effect relations between events using logical clocks [25]. Asynchronous systems must also take into account the FLP impossibility result [26], which states that there is no deterministic algorithm that reaches consensus in an asynchronous network when in the presence of at least one faulty node. Several mechanisms have been studied to circumvent this impossibility while retaining an asynchronous system, some of the more popular ones being the use of randomization algorithms for the convergence of consensus values and the termination of consensus instances [10].

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

Partially Synchronous Systems. Partial synchrony is derived from the fact that while real-world systems may observe some time bounds (most of the time), these timing assumptions are not guaranteed, creating periods of time where the network is asynchronous. The partial synchrony approach is designed around the fact that the network will correctly hold the proper timing assumptions eventually, just that it is not known when. This makes it so execution is dictated in an eventually synchronous approach, where once it reaches a long enough synchronous period of time, the system will be able to hold timing assumptions, allowing for the remainder of the execution to terminate or do something useful towards system progress. This approach was first introduced by Dwork, Lynch, and Stockmeyer (DLS) [27] to circumvent the FLP impossibility, as a protocol that maintains safety during asynchronous periods will be able to guarantee termination once it reaches a period of synchronous behaviour. These systems are usually complemented with an *eventually perfect failure detector*, which can adjust and leverage the timing assumptions in eventually synchronous systems in order to correctly suspect faulty participants after an unknown amount of time.

4.2.2 Communication Patterns

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

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

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

Tree-based. Tree-based patterns are an extension of One-to-All patterns, where to reduce the load of a singular aggregation and dissemination coordinator, the system instead divides itself into multiple layers of aggregation and dissemination. The root of the tree still executes the functions of a leader, but instead of relaying messages to all other processes, it only does so to its children nodes. Afterwards, each child can be the parent of another set of nodes, performing first the dissemination of information to its children and later the aggregation of the gathered responses to send to its own parent. Thus, this structure spanning a tree spreads the load and responsibility of One-to-All communication amongst all participants charged with the role of parent node, at the cost of further increasing the number of communication steps in each consensus phase. The added latency can be accounted and compensated for by a variety of mechanisms, such as pipelining techniques, which we describe in more detail in Section 4.3.

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

4.3 Pipelining

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

The application of pipelining is dependent on how often a system may face reconfiguration. In protocols based on a stable leader policy, a designated leader will carry out pipelining tasks throughout multiple instances until it is suspected faulty. The execution of a view-change breaks the pipeline, requiring the following leader to resume it once elected. In the case of rotating leader-based protocols such as Chained HotStuff [3], as each instance is assigned to a different leader, pipelining is performed by making it so each leader executes and certifies only one phase of the protocol and then passes the pipelined information to the following leader. This method of ensuring leader fairness in rotating leader pipelined protocols is often referred to as Leader-Speaks-Once (LSO) [28]. This is facilitated by the fact that HotStuff leverages *Quorum Certificates* (QC), which are used as a collection of votes from a quorum in regard to a phase of consensus. That said, QCs can be used to aggregate and disseminate information, and, in the case of Chained HotStuff, generic QCs can be utilized to aggregate information related to concurrent rounds of different consensus instances, simplifying the transfer process of all the necessary information to the subsequent leaders in the pipeline. However, the LSO pipeline approach brings forth liveness concerns due to the existence of the Consecutive Honest Leader (CHL) property [28], which can be summarized as the fact that to reach the successful commit of a single proposal, then there needs to be enough consecutive correct leaders in the pipeline to execute the consensus instance from start to finish. In the case of Chained HotStuff, liveness is hindered by the fact it needs 4 consecutive correct leaders to successfully decide on consensus instances, but pipelining is still proven to be beneficial to the system's overall throughput.

4.4 Reconfiguration

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

4.4.1 Leader-based Reconfiguration

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

Stable Leader Policy. In a Stable Leader Policy setting, the actuating leader is only switched when enough nodes in the system suspect that it is faulty. Once a quorum of nodes deems that the leader is showing arbitrary behaviour, they trigger the reconfiguration mechanism through the exchange of messages to guarantee that the next instance of consensus is run with a new leader. An example of this would be in PBFT [2], where nodes broadcast to all other nodes a *VIEW-CHANGE* message if the current leader is not making enough progress within a given time bound. Stable Leader based systems can leverage this reconfiguration policy to drive higher transaction throughputs in the system as, once a correct leader is picked, it means that the system can avoid the need for reconfiguration for long periods of time in most scenarios. However, in the context of blockchains, this type of behaviour is undesirable as maintaining the same leader for long periods of time raises questions about leader fairness and transaction censorship, as a biased leader can select client requests to be excluded from block proposals for indefinite periods of time. Lastly, byzantine leaders can greatly diminish the throughput gains of stable leader systems by acting correctly within the bounds of the reconfiguration trigger, at an intentionally slower pace [10]. Such cases require that protocols adopt more fine-tuned and inquisitive fault detection mechanisms.

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

4.4.2 Reconfiguration Complexity

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

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

For systems based on One-to-All communication such as HotStuff, processes that deem a view change necessary send a NEW-VIEW message directly to the next view's leader. Once this node has aggregated a quorum of NEW-VIEW messages, it can effectively broadcast the collected votes and the necessary system state so that the protocol's next consensus instance runs in the new view. In HotStuff's case, all nodes send a NEW-VIEW message between consensus instances to the next leader as dictated by its Rotating Leader policy. Overall, this approach has a linear communication complexity, proving to be a more scalable alternative in systems that already leverage a coordinator mechanism. This is the subject of study by solutions such as Cogsworth [29], which aim to make the optimal time of leader-based reconfiguration more common through the use of speedup mechanisms.

4.4.3 Reconfiguration Goal

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

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

In short, a robust tree can be seen as a logical tree where neither the root nor any internal node is faulty. This is a strict assumption that excludes cases where a tree may lead to consensus even if not robust: for example, trees whose faulty internal nodes have all their children faulty as well. Inherently, the amount of possible tree configurations is considerable in size, while only a small portion of said configurations are robust in nature. As a solution, Kauri leverages this definition to design a reconfiguration mechanism that attempts to find a robust tree configuration in under t attempts, falling back to a star topology in case of failure. This is done by framing the reconfiguration problem as follows: by modelling a tree configuration as a static graph and the sequence of trees as an *evolving graph*, we can guarantee that there will be a robust static graph under the definition of *recurringly robust evolving graph* [4]:

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

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

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

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

4.5 Systems of Note

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

4.5.1 PBFT

Being one of the more well-known BFT algorithms, Practical Byzantine Fault Tolerance (PBFT) was the first protocol to enable systems to tolerate byzantine faults in a non-synchronous environment. More specifically, while PBFT's safety guarantee is upheld in an asynchronous environment, PBFT still requires at least a partially synchronous network to guarantee liveness [15]. Based on an All-to-All communication pattern alongside a leader-based design, PBFT reaches consensus in three communication steps. While this design may provide little scalability and high network saturation, PBFT is still one of the most robust BFT protocol definitions and is often used as a building block towards the different solution requirements of a variety of blockchain consensus problems. Protocol execution is as follows (Figure 1):

Clients first broadcast their request to all processes in the system. The currently assigned leader process, upon receiving a client request, communicates it by broadcasting a *PRE-PREPARE* message to all processes. This proposal message effectively orders the client request by assigning it a sequence number. Additionally, it contains the view it is being sent in and the digest of the request. PRE-PREPARE messages are kept small in size as they are used as proof that the request was assigned a sequence number in the specified view during the execution of a view-change procedure. All nodes upon receiving the PRE-PREPARE message acknowledge it by replying with a *PREPARE* message bearing similar contents, which is broadcast to all nodes. When any node receives a byzantine quorum of $2f + 1$

matching PREPARE messages, it will then proceed with the broadcast of a *COMMIT* message to all nodes once again. Now, when a process receives a byzantine quorum of COMMIT messages, it will know that consensus was reached as there is a sufficient amount of correct replicas that have also reached the same conclusion. Thus, it can be declared that the client request has been collectively *decided* on and ordered with the sequence number it was assigned in the PRE-PREPARE phase by the leader.

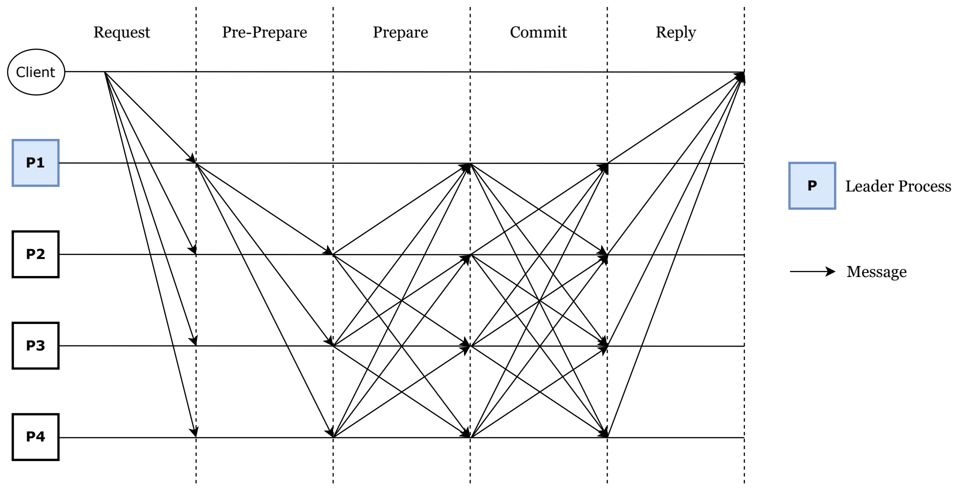

Figure 1: PBFT Normal Case Execution

To guarantee liveness, PBFT implements a view-change mechanism to enable progress when the leader process displays faulty behaviour. In PBFT, views are incrementally numbered, and no two consecutive views have the same leader. The leader-to-view assignment follows a looping pattern, given by $p = v \ mod \ N$, where p is the id of the leader node, v is the number of the view and N is the number of nodes in the system. View changes are typically initiated through the use of a timeout mechanism, where a timer is initialized when processes receive a new request. If the system does not make enough progress within the defined timeout interval, the system first attempts to stay within the same view by doubling the timeout value. If it fails to make progress once more afterwards, then each of the participant's timer is triggered, signalling each process to execute the view change protocol. Just like normal case operations, the view change protocol presents a communication complexity of $O(n^2)$, alongside the added latency invoked by the timeout system, making view changes very costly. Additionally, it has been noted that attackers can collude to arbitrarily delay the system's performance once the timeout reaches a big enough value, making it so only certain client requests are processed as late as possible [10]. To reduce the impact of view changes in system performance, certain algorithms have been proposed to enable broadcast All-to-All systems like PBFT to more commonly reach a theoretical $O(n)$ optimal reconfiguration time, such as the leader-based view change protocol proposed by Naor *et al.* [29].

4.5.2 HotStuff

To promote efficiency and scalability, HotStuff builds on top of PBFT. This is done by utilizing the leader to aggregate protocol votes and disseminate results, having a One-to-All communication pattern combined with threshold signatures to guarantee linear communication. The goal of HotStuff is to create a protocol based on *optimistic responsiveness* [30] combined with linear communication to reduce the message complexity of not only normal case operations but also view change sequences. For a protocol to be responsive, it means that it can reach consensus in time that is dependant only on the actual message delays instead of being dependant on any known upper bound on message transmission delays [31], the latter being a frequent case in systems based on eventually synchronous models.

To compensate for the heavy workload the leader node faces and to also promote what is often referred to as *chain quality* [32], HotStuff makes it so each consecutive instance of consensus initiates a new view with a new leader from its node rotation, diverging from PBFT's stable leader policy. While this rotating leader approach improves fairness and load balancing, it also creates a time overhead on protocol execution due to the weight and frequency of leader node election [23]. Additionally, on the worst-case of f cascading faulty leaders during leader rotation, the system may be forced to take $O(f * n)$ rounds to terminate a consensus instance (which is on the magnitude of $O(n^2)$ complexity [15]). Other than the rotating leader behaviour, protocol execution follows PBFT's design, with the additional detail of each phase requiring both the aggregation and dissemination of messages through the leader, as previously described for One-to-All systems in Section 4.2.2. Thus, basic HotStuff execution can be expressed by Figure 2.

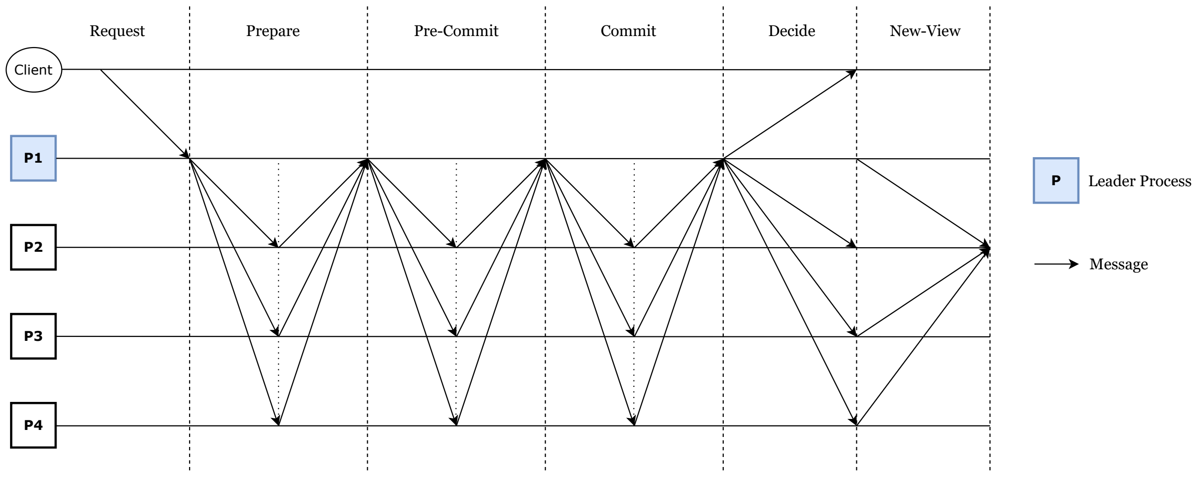

Figure 2: HotStuff Normal Case Execution. At the end of the consensus instance, processes send a NEW-VIEW message to the leader of the following consensus instance.

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

4.5.3 Kauri

To address the leader bottleneck found in One-to-All systems like HotStuff, Kauri shapes the system's communication pattern into a tree topology with multiple layers of aggregation and dissemination. This way, the workload assigned to the leader node is spread across all nodes that belong to the inner processes of the tree, which execute similar functions to the root except at a sub-tree level. When we say that a node has a fanout of m, it means that it has m children nodes which it is assigned to disseminate messages to and aggregate the votes of. In the end, this tree structure incentivizes load balancing and improves system scalability, with the trade-off of creating additional communication steps for each layer of depth that the tree has. The protocol's execution can be seen on Figure 3. Kauri divides the execution time of a node during a round of consensus into three categories:

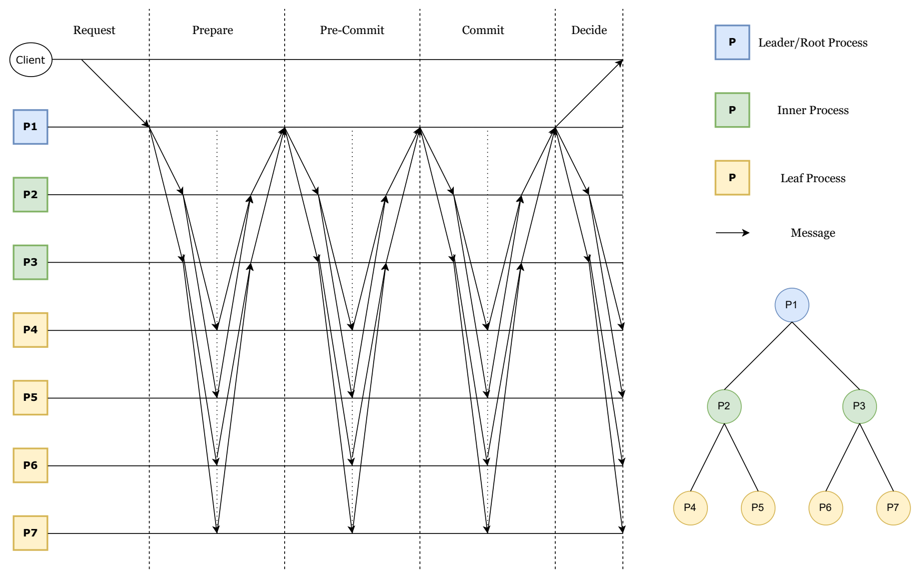

Figure 3: Kauri Normal Case Execution with a balanced tree of N=7 nodes.

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

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

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

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

It is important to refer to the fact that the issue of implementing topological awareness in Kauri has been previously discussed in a similar work [33]. The aforementioned solution proposed an optimistic clustering mechanism based on latencies, with the aim to exploit the local communication benefits of a balanced tree with n balanced clusters branching from the root. It then leverages this topology-aware clustering approach to implement a rotating leader scenario to provide the necessary chain quality features to Kauri in a WAN deployment.

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

4.5.4 GeoBFT

GeoBFT is a partially synchronous BFT protocol designed to handle geo-scale deployments alongside the ResilientDB permissioned blockchain [34]. The aim is to provide both high scalability and high decentralization benefits to blockchain systems without compromising the throughput of the system. This is achieved through the topologically aware grouping of nodes into local clusters, which lets the protocol distinguish between local and global communication towards a more efficient consensus design. GeoBFT optimistically allows each cluster to make decisions independently and only afterwards relays their decisions through a global channel, thereby globally sharing and ordering all of the locally decided-upon client requests. Effectively, this means that an instance of consensus is separated into three stages: local replication, global sharing and ordering and execution. All of these stages rely on the fact that each cluster has a coordinator named the primary replica. This primary performs the role of leader for local PBFT replication but also performs the role of intermediary with the other primary replicas of the remaining clusters during the global sharing phase.

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

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

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

4.5.5 Fireplug

Fireplug presents a flexible architecture model for building SMR for geo-replicated databases. Although not initially presented in the context of blockchains, Fireplug proves to be of interest as it tackles both the requisites for scalable BFT consensus and how systems may leverage different kinds of heterogeneity towards their overall robustness to compromises. Fireplug implements geo-scale distribution by defining the existence of datacenters and leveraging what it defines as a hierarchical composition of multiple instances of BFT-SMART [35]. This way, each datacenter communicates locally with a different instance of BFT-SMART and multiple datacenters may communicate with each other through a global instance of BFT-SMART. Furthermore, Fireplug abstracts away the interface to interact with each database by implementing both a proxy for multi-versioned data and a replication middleware at a node-based level. This heterogeneity abstraction, coupled with the software diversity provided by its N-Version Programming support, enables Fireplug to leverage diversification at an inter and intra-datacenter scope. This helps reduce the probability of multiple replicas being compromised simultaneously through the same vulnerability, whether at a technology or software level.

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

4.5.6 Mir-BFT

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

To enable the correct ordering of client requests even when there are simultaneous proposals occurring, Mir-BFT multiplexes multiple instances of its broadcast primitive over a partitioned domain of client requests, which can guarantee liveness and safety properties but also implement data duplication prevention mechanisms. The assignment of leaders to the partitioned client hash space is rotated over the course of consecutive *epochs*, meaning that the transition between them can be seen as a view-change in the system. Some of the limitations of Mir-BFT include the high communication costs of the broadcast-based behaviour of the system and the high complexity of the design [23]. An example of the latter is how the batching of client requests depends on concurrent request handling data structures, which is a complex component with a lot of discussion regarding reliable implementation.

4.5.7 DBFT

DBFT is a leaderless protocol that runs on a partially synchronous network model. In this case, the scalability, load distribution and throughput gains all come from the fact that all nodes play the same role in the execution of consensus, while the offered decentralization avoids bottlenecks. To avoid the FLP impossibility, DBFT utilizes the notion of a *weak coordinator*, whose goal is to help the algorithm terminate when non-faulty nodes know that their proposals might be decided. This goes against the habitual coordinator behaviour of leader-based systems, where the leader imposes their value on other nodes and forces the system to wait for their decision. It is through this weak coordinator that the system is able to resolve conflicts related to its concurrent proposals. However, for this conflict resolver to work, it requires the existence of partial synchronicity, which is a drawback to the leaderless design of DBFT, as other state-of-the-art leaderless protocols such as HoneyBadgerBFT [36] are able to execute consensus in asynchronous environments.

It was recently argued that this type of leaderless design can drive high throughput and ensure good scalability in WAN settings [14]. This was alleged as DBFT's parallel distinct proposals implement the notion of cumulative All-to-All broadcasts. Through this, the system is able to drive a high payload transfer rate (*goodput*) and exploit larger proposal batch sizes to avoid the waiting time between broadcasts (*hiccup*).

4.5.8 MyTumbler

MyTumbler is a recent timestamp-based leaderless BFT protocol that is able to execute in an asynchronous network model. It aims to create a system that is able to move at the speed of the network delay, where commits are either proposed or aborted, instead of being dependent on the speed of coordinators and the difficulties of tuning the maximum network delay in eventually synchronous systems, where an improperly set timeout bound may lead to excessive consecutive leader changes or too big of a delay to recover from faults. MyTumbler additionally implements the notion of Super Multi-value Agreement (SuperMA) to leverage an optimal fast path in the randomization process often found in asynchronous protocols. By utilizing a promise mechanism, the non-deterministic common coin [11] component of randomized consensus may be sped up if a quorum of correct processes has submitted the same value for consensus initially. Overall, MyTumbler imposes a trade-off of high communication complexity for the possibility of quick termination, alongside the possibility of transactions having to abort due to conflicts in the asynchronous network.

4.5.9 Other Systems.

We highlight two additional systems which are relevant for our solution discussion in Section 5: PrestigeBFT [17], which leverages the concept of node reputation (i.e., a metric for the perceived correctness of a participant) to build a system that keeps track of the past faultiness of its nodes, and Helena *et al.*'s attempt at a topologically aware Kauri [33], whose solution was analysed more in-depth in our discussion of Kauri above.

4.6 Discussion

BFT Consensus Algorithm	Coordination Approach	Communication Pattern	Topologically Aware	Leverages Pipelining	Offers $N = 3f + 1$ Resilience
PBFT [2]	Leader-based	All-to-All	✗	✗	✓
HotStuff [3]	Leader-based	One-to-All	✗	✓	✓
Kauri [4]	Leader-based	Tree-based	✗	✓	✓
GeoBFT [24]	Group-based	Hierarchical Groups	✓	✗	✗
Fireplug [6, 37]	Group-based	Hierarchical Groups	✓	✗	✗
Mir-BFT [21, 22]	Multi-leader	All-to-All	✗	✗	✓
DBFT [19]	Leaderless	All-to-All	✗	✗	✓
MyTumbler [20]	Leaderless	All-to-All	✗	✗	✓
Our Approach	Leader-based	Tree-based	✓	✓	✓

Table 1: Proposed approach when compared to other existing BFT consensus algorithms.

Table 1 summarizes the main characteristics of the state-of-the-art algorithms mentioned above. The features that our comparison aims to highlight are as follows:

Coordination Approach. The coordination approach is one of the most integral factors for dictating how a system handles the load distribution between its nodes and if a system may even need to reconfigure at runtime in the first place. The role of coordinator implicitly comes with additional processing costs and responsibilities, meaning that the system may quickly rise in complexity when it is performed by a multitude of nodes. Systems that spread the responsibility of coordination (GeoBFT, Fireplug, Mir-BFT) or remove the role of dedicated coordinators altogether (DBFT, MyTumbler) often obtain throughput gains from the added concurrency when the communication is non-redundant (distinct proposals). As a drawback, however, the system must implement more complex mechanisms that both deterministically order all concurrent transactions in the system and avoid conflicting simultaneous proposals, which can lead to unsatisfactory design solutions. It is important to note that leaderless systems such as DBFT and MyTumbler do not require the same notion of reconfiguration as the other protocols, since every node has equal capabilities and blockchain membership reconfiguration is out of the scope of their design. Our solution, in order to properly benefit from the load distribution benefits of its tree topology and maintain an easily scalable design, should aim to be leader-based.

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

Topologically Aware. To be topologically aware is to utilize information regarding how the network or nodes are structured to the system's advantage. The most common example of this is to attempt to use geographic localization to promote communication in faster node links. This is usually applied through the definition of a heuristic external to the protocol's execution, which enables the clustering or grouping of nodes. Such cases lead to highly decentralized designs, as seen in GeoBFT and Fireplug. Another example of topological conditions that can be leveraged for system efficiency is how computationally strong nodes are. This information allows leader-based, group-based and multi-leader protocols to assign coordination responsibilities to priority nodes which can deliver better performance to the system and mitigate some of the consequences of design bottlenecks. We can state that none of the protocols studied, with the exception of Fireplug, have this capability. Our approach should attempt to apply topology awareness to define a dynamic and well-performing reconfiguration algorithm that takes into consideration the asymmetries present in a WAN deployment.

Leverages Pipelining. Pipelining is an important factor for increasing consensus performance when in the presence of protocols with ample idle time such as Kauri. Without it, the drawbacks from the additional communication steps of their aggregation and dissemination behaviour would be too costly when compared to the performance of broadcast-based systems such as DBFT. On the other hand, pipelining is a complex mechanism which may need fine-tuning to be able to be fully taken advantage of. An example of this is with the ISS [22] version of HotStuff, where the presence of the modular multi-leader framework alongside pipelining made it so the protocol could not take full advantage of the parallelization, as part of the partitioned proposal space was wasted on dummy batches needed for the extra rounds to avoid breaking the pipeline. Our approach should leverage pipelining in a way that reconfiguration may take into account the links used in the previous configuration.

Offers $N = 3f + 1$ Resilience. This is a critical aspect of byzantine consensus, as mentioned in Section 3.2. It is important to highlight this design choice because it is the central weakness of the highly decentralized and highly scalable protocols of GeoBFT and Fireplug. Optimistically, our solution should still maintain the optimal expected resilience while attempting to benefit from the decentralization tactics applied by these two designs. It is important to also highlight that in the case of Kauri, we consider that it has $N = 3f + 1$ resilience due to the fact it exclusively utilizes robust trees for configurations. Internal node faults in Kauri create faulty underlying sub-trees due to the aggregation and dissemination behaviour displayed by its topological tree. This makes it so assessing correct behaviour with different quantities and arrangements of faulty nodes in Kauri configurations a very complex and difficult task.

5 Approach

In this section, we outline an approach that follows the criteria that we have established in Section 4 and explain how it will be used towards the goals defined in Section 2.

We aim to enhance Kauri with a reconfiguration mechanism that not only utilizes a more dynamic approach to create trees with reduced latency when compared to the original protocol but also avoids the need to globally reconfigure trees when a fault may be able to be fixed locally within a sub-tree. The approach, when combined with a component that provides heuristics regarding the topological characteristics of the system's WAN deployment and the reputation of its nodes, will be able to more efficiently reconfigure with a top-down procedure that gives priority to more desirable and trusted participants. For our approach, for example, we can assume the existence of two modules: one that is able to provide the system with metrics regarding its network infrastructure (e.g., Vivaldi [38]) and another one that is able to compute and keep track of node reputation (as exemplified in works such as PrestigeBFT [17]). These modules can then provide the system with up-to-date information that can be easily be fetched during the reconfiguration procedure, allowing the reconfiguration mechanism to weigh which characteristics it deems more important for system performance.

This approach is more likely to provide the system with efficient and robust trees in stable leader scenarios, with the possibility of partial reconfiguration. In a partial reconfiguration context, we limit the tree's reconfiguration to only a sub-tree within the encompassing tree, allowing us to maintain a majority of nodes unchanged from the last configuration. Through this concept, we may be able to drive progress in the system throughout the partial reconfiguration procedure while also avoiding the risk of breaking robust sub-trees that weren't deemed faulty. Additionally, by achieving a topologically efficient tree structure, we are able to decrease the latency and increase the throughput of normal-case operations. The way the heuristic-based assignment component obtains and leverages its metrics will be abstracted away in our discussion as it is not within the scope of this project.

5.1 Challenges

Given the previously discussed attempt at a more topologically aware Kauri in Helena *et al.*'s work [33], we shift our attention from the topology heuristic definition and clustering process and instead focus on the design of a reconfiguration mechanism that is already topologically aware through abstracted away means. By obtaining both the reputation of nodes and topological features of the system through encapsulated modules, our approach can instead focus on how we can use this information to benefit the system. An example of an application process that leverages this information can be seen down below in Section 5.2.2. That said, when dealing with a paradigm such as reconfiguration, we have three major concerns that we have to tackle:

- How does the protocol know what needs reconfiguring and when?
- How does the protocol establish a feasible goal for the reconfiguration?
- How complex is the process of reconfiguration towards the established goal?

We can answer the previous questions as follows.

First, we need to define the two reconfiguration scenarios in our system: the case of *full reconfiguration*, where the root node is suspected faulty, and the case of *partial reconfiguration*, where the parent node of any sub-tree is suspected faulty (excluding the case of the root). In both cases, the information exchanged must be global to all the nodes in the system, so as to not open the opportunity of byzantine nodes excluding information to certain nodes. In the case of full reconfiguration, a quorum of all the nodes in the system can elect to change the leader if it is deemed that the system is not making enough progress (hence, the leader is suspected faulty). In the case of partial reconfiguration, the parent of any sub-tree may order its children to rearrange if the system deems that their specified sub-tree is not making any progress.

In the second scenario, it is important to note that we aim to give power to parent nodes and not children nodes, as doing it the other way around may lead to undesirable risks. To put it simply, the case of a byzantine parent will always be the worst-case scenario for a sub-tree, meaning that if a byzantine parent orders its sub-tree to rearrange, it will not provide a more harmful outcome to the system other than the delay experienced by the reconfiguration itself. Meanwhile, if we took an approach where the children nodes were able to obtain a quorum that could force the sub-tree to be rearranged, a correct parent could be forced out of its position by a quorum of colluding malicious participants, leading to a worse system state. This situation is possible within the scope of any sub-tree, as there can be a majority of byzantine nodes colluding at this level. While this latter approach opens up the possibility of correct children nodes being able to force a byzantine parent out of its position, it equally opens up the risk of a correct parent being kicked out of its inner node position, which is undesirable. In the following section, we will discuss two possible approaches to initializing both full and partial reconfiguration scenarios.

Secondly, we need to make it so that reconfiguration is a deterministic process. By leveraging the concept of complaint messages, which will be our reconfiguration requests with information regarding which sub-trees are failing and which nodes are suspected, participants will be able to simulate the reconfiguration process (i.e., accurately predict the results for the different steps of the reconfiguration given a set of complaint messages) through the use of a deterministic function in order to verify a desirable outcome for reconfiguration. This enables a node to propose a deterministic goal by utilizing a set of complaint messages as a foundation. Additionally, any other participant can confirm the validity of that goal by taking into account i) the current tree and ii) the proposed goal's set of complaints. Thus, reconfiguration can be split into two phases: the *election* phase, where nodes propose their goals alongside their respective complaint sets, and the *voting* phase, where all nodes process and validate the candidate proposals before casting their votes accordingly.

To implement the aforementioned deterministic process, we will leverage the notion of Recurringly Robust Evolving Graph introduced by Kauri and referenced in Definition 6. Through the use of a deterministic function, we can simulate any reconfiguration process when taking our evolving graph $\mathcal{G}$, our current static graph equivalent to our current tree $\mathcal{T}_C$, a set of complaint messages $\mathcal{C}$ and our abstracted assignment component into consideration.

Lastly, as our approach will follow the behaviour of a top-down tree algorithm (i.e., reconfiguration starts at the highest layer parent and moves down recursively to their children), it is easy to assume that it may reach the established goal in h parent-to-children communication steps, where h is the height of the sub-tree being reconfigured. However, as the system state must be reliably spread across all nodes, it is important to note that reconfiguration requires broadcast-based communication patterns that include all nodes in the system. As a starting point, our approach utilizes this broadcast-based behaviour with the possibility for optimisations in the future. The system itself must also be halted during the reconfiguration process, both in the case of full reconfiguration and in the case where the system is not able to obtain a quorum of correct nodes during partial reconfiguration, meaning that there is no progress during this time.

5.2 Provisional Approach

To follow up on the high-level approach defined in Section 5 and the challenges that have been discussed in Section 5.1, we propose a preliminary algorithm that is described below. This proposal will be split into two parts: i) the reconfiguration trigger, which will have two proposed mechanisms and ii) the top-down reconfiguration execution.

5.2.1 Reconfiguration Trigger

As previously mentioned, this mechanism will be leveraging the notion of *complaint* messages, which will contain information that can be useful to infer which sub-trees are failing and which nodes are suspected for said faults. These complaint messages contain complaint reports that will infer if progress is being hindered by a lack of messages from the parent node or by a lack of replies from its children nodes, from the perspective of the reporter node. To exemplify, as can be seen in Figure 4, complaint reports from nodes 1, 6 and 7 may prove to be key factors in identifying node 3 as faulty when given the context that the root's leftmost sub-tree is not making any progress. That said, once compiled together, quorums of

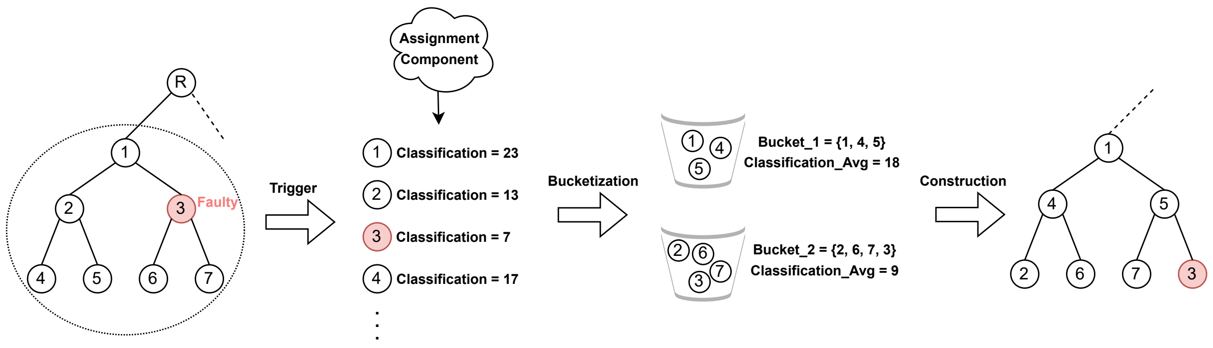

Figure 4: Partial Reconfiguration example for a sub-tree with N=7 nodes where I=3 are inner nodes.

complaint reports can establish intersections in the root cause of arbitrary behaviour and the candidate suspect nodes behind it. The process of creating, communicating and asserting complaints in the system is by far the more complex topic in this provisional approach, leading us to propose two alternative approaches:

Global Complaint. In this approach, once a node suspects it is in a sub-tree which is not making enough progress within a defined time bound, it communicates a complaint message containing a complaint report to the entire system through a reliable broadcast channel. This way, the system can essentially be aware as a whole of the reported complaints, making sure that complaints are sent globally and that every node has the opportunity to receive the same complaints. Once a big enough quorum of complaint messages are propagated in the system, each node that possesses a quorum of complaints is able to determine and propose a reconfiguration goal in what we previously referred to as the *election* phase. Inherently, this proposal must exhibit the set of complaint messages used to generate it, to enable voters to check its validity. This means that all nodes can run a deterministic function which will be able to process the elected sets of complaint reports and establish the best candidate reconfiguration goal. The decided-upon goal is then reliably broadcast to all nodes in the system, and with a byzantine quorum of equivalent messages, the system can begin the execution of the top-down reconfiguration to enter a new view with a new configuration.

Restricted Complaint. This approach relies on processes sending their complaints to select participants instead of attempting to send them to the entirety of the system. Different recipient selection policies may be used, depending on the context of the sender and the contents of the complaint. If the system is not able to progress within a defined time bound or if a node reaches a quorum of complaints and attempts to change the current tree, we initiate the *election* phase. Just as how it was described in the global complaint approach, during the election phase, all nodes that have received complaints communicate their reconfiguration goal alongside the set of complaints used to generate it. However, in this case, elected participants will act as the leader of the sub-tree they aim to reconfigure. Participants are then once again able to deterministically vote for the best candidate reconfiguration goal by validating and processing the proposed complaint sets. Votes are sent directly to the participant's chosen candidate leader, which may start the top-down reconfiguration execution once it receives a quorum of votes.

5.2.2 Top-Down Execution

Once reconfiguration has been triggered and its goal has been established, we know which sub-tree will be reconfigured. Additionally, in the case it is a full reconfiguration, we also know which node has been selected to be the leader of the new view. The reconfiguration process begins by assigning all of the sub-tree's nodes with a value calculated by a higher-level component. This value can be seen as a metric which is the combination of the node's topological qualities (in the context of the sub-tree) and the node's reputation, updated with the recently inferred activity from the accumulated complaints. To simplify, we will refer to this value as classification, where a higher classification means a more desirable node.

After the assignment process, we can utilize the same bucketization strategy that Kauri employs: we distribute all nodes along t disjoint bins, each of size equal to or greater than the total number of internal nodes in the sub-tree. The difference is that in this case, the buckets are filled progressively and in order, starting from the higher classification nodes to the lower classification ones. After the buckets are filled, we can average out the classification of its constituent nodes and add a weight representing the average perceived latencies between themselves. With this, each bucket will now have an assigned classification, and the system can attempt to establish a robust tree with the currently generated buckets. It is also important to note that the buckets internally have their nodes sorted from highest classified to lowest classified.

With the buckets decided, tree construction ensues. To guarantee a deterministic pick order, the ordering for each bucket is decided on by the leader and subsequently passed down recursively to its children. The process is as follows: the root of the sub-tree picks the m leftmost nodes in the bucket (highest classified). This selection process is repeated for each of its children, starting from the leftmost child to the rightmost child, and then afterwards to their children recursively. Once the bucket is emptied (meaning that the last internal nodes have been picked), the leaf nodes are decided with the same selection policy starting from the next highest classified bucket till the last least classified bucket. With this in mind, we can simplify the communication in a top-down approach: the root sends a certified message directly to its newly assigned children nodes to validate their position. This message includes the generated buckets and the receiving node's position in the sub-tree. This means that upon its reception, a node knows which participants they will have to contact to assign them as their children. Additionally, when a node receives an assignment message, it certifies its validity and subsequently signs it to forward it to its children. By forwarding their parent's message, every node is able to prove it was correctly selected for its current position in the sub-tree.

A high-level example of the reconfiguration process can be seen in Figure 4: after processing a quorum of complaint messages and deciding on a goal, the system dictated that the sub-tree to the left of the root node was not making enough progress, thus initiating partial reconfiguration. In this case, the root is able to command the sub-tree to update its node classification values through the assignment component, which leverages the modules referred to in Section 5. As previously stated, a higher classification value means that a node is more desirable for reconfiguration, meaning that it offers better topological characteristics, is more trustworthy, or both. It is of note that the values attributed in this example are arbitrary, with the exception of Node 3, which has a higher probability of having a lower classification due to it being suspected in the complaints that triggered reconfiguration. Afterwards, as we know that there are $I = 3$ inner nodes in total and the tree has a fanout of $m = 2$, we create $t = 2$ disjoint bins each of size ≥ 3. As this sub-tree only presents $f = 1$ faulty nodes, we know through Def. 7 that since $f < m$ we are able to obtain at least one bucket that has no faulty nodes in it. Thus, this bucketization strategy lets us pick a bucket to fill in all the inner node positions of the sub-tree with correct nodes, while simultaneously increasing the probability of it being the first bucket through the use of its classification mechanism. Finally, partial reconfiguration is able to construct an efficient and robust tree for the suspected sub-tree.

5.2.3 Discussion

It is safe to say that the efficiency of our solution will be heavily dependent on the added cost of having two phases for the reconfiguration goal selection and also on the algorithm chosen for the trigger mechanism. While the global complaint approach may lead to a more straightforward solution, the overhead brought by its higher communication complexity might diminish the expected results on executions with higher numbers of nodes. Meanwhile, the restricted complaint approach, while lighter on communication complexity, may be difficult to implement at a partial reconfiguration level, requiring careful optimisations in both its selection policy and the election voting phase. The objective of specifying a complaint system as part of the solution is directly linked with the need to avoid byzantine nodes in the internal positions of the tree, which should be a top priority if we aim to avoid situations where we have to reduce the tree to a star topology. By utilizing a complaint system, we are able to provide more up-to-date information regarding the behaviour of nodes from the current configuration, which could later be put to use by our classification assignment component.

On another note, the top-down reconfiguration in and of itself shows promise in regards to efficient leveraging of system heterogeneity towards well-performing dynamic trees. Its main hindrance is how dependent it is on the quality and efficiency of its abstracted classification assignment component. However, on a theoretical implementation that can take into account different system features such as node reputation, node link bandwidth, node latency, node capacity, and pipelining potential, we are able to provide a good baseline heuristic to designate nodes that are both trustworthy and reliable.

6 Experimental Evaluation

The purpose of this section is to describe an evaluation plan that will take into account the characteristics defined in our provisional approach in Section 5 and the goals outlined in Section 2.

That said, by promoting the creation of topologically aware trees in WAN environments, we aim to increase both the efficiency and scalability of Kauri. To that effect, it will be important to measure the performance of the new trees when compared to Kauri's old reconfiguration approach. A variety of real-world contexts may be used, including systems with varying node quantities, different geographic deployment coverages (which consequently induce different bandwidth strengths), different block proposal sizes, and varying quantities of node faults. Through comparison in these contexts, we will be able to evaluate the added performance of our solution in both normal-case operations and throughout the reconfiguration process, including partial and full reconfiguration scenarios. Additionally, it is important that we pit our reconfiguration trigger algorithms against one another to finalize which one would be most beneficial to our solution. The performance metrics we aim to analyse for our evaluation are as follows:

Latency. Latency is one of the key factors that can be analysed to infer the scalability of our solution. It is evaluated by measuring the time it takes for the system to finish a consensus instance from start to finish and can be compared directly with other systems (most importantly, with Kauri).

Throughput. Throughput is another metric that can be analysed to infer the scalability of a system. It is usually measured by counting the number of client requests decided on per unit of time in the system (kreq/s). Throughput can be used to more accurately analyse the impact of design choices and evaluate in which settings a system has its best performance.

Reconfiguration Speed. We can evaluate reconfiguration speed in two different ways: i) it can be the time it takes for the system to reconfigure since the moment faults are inserted or ii) it can be the time it takes for reconfiguration to terminate since the moment it is triggered. While i) takes into account both the time it takes for the system to detect faults and initialize the reconfiguration process, ii) can more easily infer the added costs of our reconfiguration mechanism's complexity, which did not exist in Kauri's original randomization strategy. Both notions are valuable assets when comparing which reconfiguration approach provides a more reliable solution. These metrics can also be used to compare the execution times of partial and full system reconfigurations.

Reconfiguration Success Rate. To deduce if the concept of partial reconfiguration brings any notable benefits to protocol design, it is important to measure how often partial reconfiguration succeeds at creating robust trees. Although a probabilistic metric, with sufficient repeated testing in different test scenarios, we may be able to infer if partial reconfiguration achieves robust trees at an equal or better rate when compared to full reconfiguration (the desirable case). This metric can also be discussed with the speed of reconfiguration in mind and the added benefits of maintaining a portion of the tree's previous node positions in the case of partial reconfiguration.

As mentioned above, when compared to Kauri's original solution, this new version of the reconfiguration procedure has an added cost. To measure if this additional cost on reconfiguration is justifiable, we will first have to measure our new approach in the context of Kauri's original benchmarks. This means we must directly compare tests where the bandwidth, CPU capacity and workload of nodes are the exact same as Kauri's, as to correctly infer the new mechanism's performance.

7 Schedule of Future Work

The proposed schedule for future work is presented in Figure 2.

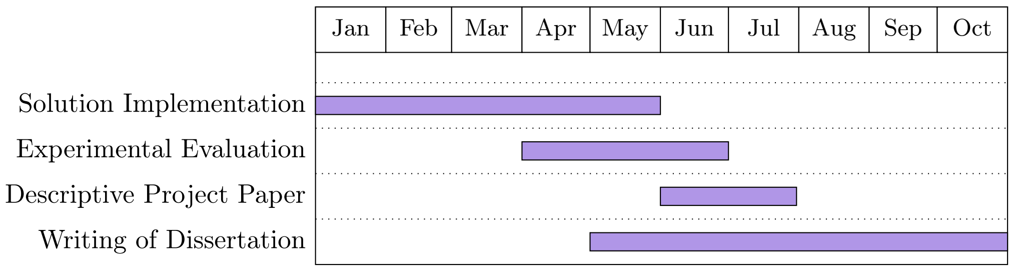

Table 2: Future Work Schedule

8 Conclusion

In the realm of permissioned blockchains, multiple approaches have been discussed to design scalable BFT consensus protocols for systems with elevated numbers of participants. While Kauri showed promise by employing a scalable tree design which leveraged pipelining techniques, it encountered challenges in real-world applications due to its design being tailored for homogeneous settings. This was the main motivation for our work, which aims to provide Kauri with more topology-aware capabilities in WAN environments, allowing for better scalability potential. As this issue tackled the tree creation and reconfiguration process, it also opened up the possibility of partial/local reconfiguration, a feature commonly used in more decentralized and scalable protocols such as GeoBFT and Fireplug.

Having surveyed the work of several BFT protocols, we were able to establish which design patterns and features were most beneficial to solve our problem. With the goal of using topological information to Kauri's benefit, we have designed a provisional approach which can theoretically achieve better scalability guarantees in WAN settings and promote higher reliability in heterogeneous networks. Lastly, we provided an evaluation method for our proposed solution and the scheduling of future work that requires completion.

Acknowledgments. This work was partially supported by project DACOMICO (via OE with ref. PTDC/CCI-COM/2156/2021).

References

[1] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *Concurrency: the works of leslie lamport.* 2019, pp. 203–226.

[2] Miguel Castro, Barbara Liskov, et al. "Practical byzantine fault tolerance". In: *OsDI*. Vol. 99. 1999. 1999, pp. 173–186.

[3] Maofan Yin et al. "HotStuff: BFT consensus with linearity and responsiveness". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, pp. 347–356.

[4] Ray Neiheiser, Miguel Matos, and Luís 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] Marko Vukolić. "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication". In: *Open Problems in Network Security: IFIP WG 11.4 International Workshop, iNetSec 2015, Zurich, Switzerland, October 29, 2015, Revised Selected Papers*. Springer. 2016, pp. 112–125.

[6] Ray Neiheiser et al. "Fireplug: Efficient and Robust Geo-Replication of Graph Databases". In: *IEEE Transactions on Parallel and Distributed Systems* 31.8 (2020), pp. 1942–1953. DOI: 10.1109/TPDS.2020.2981019.

[7] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: *Decentralized business review* (2008).

[8] Brian Neil Levine, Clay Shields, and N Boris Margolin. "A survey of solutions to the sybil attack". In: *University of Massachusetts Amherst, Amherst, MA* 7 (2006), p. 224.

[9] Sunoo Park et al. "Spacemint: A cryptocurrency based on proofs of space". In: *Cryptology ePrint Archive* (2015).

[10] Xin Wang et al. "Bft in blockchains: From protocols to use cases". In: *ACM Computing Surveys (CSUR)* 54.10s (2022), pp. 1–37.

[11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[12] Yair Amir et al. "Steward: Scaling byzantine fault-tolerant replication to wide area networks". In: *IEEE Transactions on Dependable and Secure Computing* 7.1 (2008), pp. 80–93.

[13] Catalonia-Spain Barcelona. "Mencius: building efficient replicated state machines for WANs". In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. 2008.

[14] Gauthier Voron and Vincent Gramoli. "Planetary Scale Byzantine Consensus". In: *Proceedings of the 5th workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems*. 2023, pp. 1–6.

[15] Gengrui Zhang et al. "Reaching consensus in the byzantine empire: A comprehensive review of bft consensus algorithms". In: *arXiv preprint arXiv:2204.03181* (2022).

[16] Karolos Antoniadis et al. "Leaderless consensus". In: *Journal of Parallel and Distributed Computing* 176 (2023), pp. 95–113.

[17] Gengrui Zhang et al. "PrestigeBFT: Revolutionizing View Changes in BFT Consensus Algorithms with Reputation Mechanisms". In: *arXiv preprint arXiv:2307.08154* (2023).

[18] Giorgos Tsimos et al. "Hammerhead: Leader reputation for dynamic scheduling". In: *arXiv preprint arXiv:2309.12713* (2023).

[19] Tyler Crain et al. "Dbft: Efficient leaderless byzantine consensus and its application to blockchains". In: *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. IEEE. 2018, pp. 1–8.

[20] Shengyun Liu et al. "Flexible Advancement in Asynchronous BFT Consensus". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. 2023, pp. 264–280.

[21] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. "Mir-bft: High-throughput bft for blockchains". In: *arXiv preprint arXiv:1906.05552* (2019), p. 92.

[22] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. “State machine replication scalability made simple”. In: *Proceedings of the Seventeenth European Conference on Computer Systems.* 2022, pp. 17–33.

[23] Weiyu Zhong et al. “Byzantine Fault-Tolerant Consensus Algorithms: A Survey”. In: *Electronics* 12.18 (2023), p. 3801.

[24] Suyash Gupta et al. “Resilientdb: Global scale resilient blockchain fabric”. In: *arXiv preprint arXiv:2002.00160* (2020).

[25] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (July 1978), pp. 558–565.

[26] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. “Impossibility of distributed consensus with one faulty process”. In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[27] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. “Consensus in the presence of partial synchrony”. In: *Journal of the ACM (JACM)* 35.2 (1988), pp. 288–323.

[28] Ittai Abraham et al. “It’s not easy to relax: liveness in chained BFT protocols”. In: *arXiv preprint arXiv:2205.11652* (2022).

[29] Oded Naor et al. “Cogsworth: Byzantine view synchronization”. In: (2021).

[30] Rafael Pass and Elaine Shi. “Thunderella: Blockchains with optimistic instant confirmation”. In: *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part II 37.* Springer. 2018, pp. 3–33.

[31] Hagit Attiya et al. “Bounds on the time to reach agreement in the presence of timing uncertainty”. In: *Journal of the ACM (JACM)* 41.1 (1994), pp. 122–152.

[32] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. “The bitcoin backbone protocol: Analysis and applications”. In: *Annual international conference on the theory and applications of cryptographic techniques.* Springer. 2015, pp. 281–310.

[33] Helena Teixeira, Luıs Rodrigues, and Miguel Matos. “Arvores de Disseminaçao e Agregaçao Cientes da Topologia para Suportar Consenso Bizantino em Larga Escala”. In: *Inforum* (2023).

[34] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. “Permissioned blockchain through the looking glass: Architectural and implementation lessons learned”. In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS).* IEEE. 2020, pp. 754–764.

[35] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. “State machine replication for the masses with BFT-SMART”. In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* IEEE. 2014, pp. 355–362.

[36] Andrew Miller et al. “The honey badger of BFT protocols”. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security.* 2016, pp. 31–42.

[37] Manuel Bravo et al. “Policy-based adaptation of a byzantine fault tolerant distributed graph database”. In: *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS).* IEEE. 2018, pp. 61–71.

[38] Frank Dabek et al. “Vivaldi: A decentralized network coordinate system”. In: *ACM SIGCOMM Computer Communication Review* 34.4 (2004), pp. 15–26.