Self-adapting BFT consensus

Helena Teixeira
helena.teixeira@tecnico.ulisboa.pt

Instituto Superior Técnico
Advisors: Prof. Luís Rodrigues and Prof. Miguel Matos

Abstract. Permissioned blockchains are a class of blockchains where the processes that run consensus are limited and known by all participants. These blockchains can execute variants of *Byzantine consensus* that offer *finality*. An approach to support a large number of participants in this context is to use dissemination and aggregation trees to support the communication required to execute protocol rounds. Many Byzantine consensus protocols are leader based, and in the blockchain scenario, there are compelling reasons to rotate the leader between consecutive consensus instances, such as the distribution of the load of the leader and censorship resistance. Unfortunately, changing the tree may disrupt the pipelining of consensus instances, a strategy proposed to hide the additional latency introduced by using trees. This work addresses the challenges of implementing a rotating-leader policy in BFT consensus that uses dissemination and aggregation trees. We aim to devise strategies to select the trees used by rotating leaders that minimize the effect on pipelining and promote good load distribution among all participants.

Table of Contents

1 Introduction

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

It was recently shown that dissemination/aggregation trees could be used to support the data exchange required by blockchain consensus while distributing the load and avoiding bottlenecks [1]. However, using trees in this context increases the latency of consensus rounds. Therefore, pipelining techniques have been suggested to mitigate the impact of the increased latency in the system throughput. Kauri [1], like many other BFT protocols, is a leader-based protocol that combines these two techniques. Nevertheless, to maximize the benefits of pipelining, it uses a stable leader policy, i.e., the same leader is used for consecutive instances of consensus until this leader is suspected to be faulty. Furthermore, the same tree is used while a given leader is active. Pipelining is also used in protocols such as HotStuff[2], which relies on a star topology and starts the next consensus instance optimistically while the previous instance is still running and piggybacking messages from multiple instances on the same network packets.

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

The challenge addressed in this work is to devise a strategy to rotate the leader (and the associated dissemination/aggregation tree) with a minimal negative impact on pipelining and, at the same time, promoting a good balance of the load among all participants (it should be noted that processes that become interior nodes in a tree have more load than nodes that become leaf nodes).

The remainder of the document is structured as follows. In Section 2, we provide an overview of the Blockchain and Byzantine Consensus concepts. Section 3 discusses the related work regarding communication patterns, leader change techniques in BFT algorithms, and pipelining. Section 4 defines the preliminary proposal and the following goals. Section 5 discusses the evaluation plan and metrics. Finally, Section 6 presents the agenda of future work, and Section 7 concludes the report.

2 Background

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

2.1 Blockchain

The term *blockchain* describes the type of data storage used in Bitcoin [3] and similar systems. This report will focus on distributed ledgers that leverage the blockchain as their storage technology. A blockchain is a distributed system with the goal of building a decentralized system that provides a trustworthy service in an untrustworthy environment. The decentralized processes run a protocol to reach an agreement on a system state during normal-case operations or when there are arbitrary faults and an attack occurs [4].

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

Without trusted elements that are aware of all transactions, building a blockchain requires regular synchronization between the different participants [3]. Therefore, the primary tool for building such a distributed ledger is consensus, discussed in detail in the next section.

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

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

2.1.1 Permissionless

Permissionless blockchains are decentralized ledger platforms in which anyone can participate. It functions as an open setting. However, this means that anyone can read and write to the ledger, allowing malicious users to issue requests that try to subvert the system [5]. An adversary could easily create sufficient instances to subvert the existing consensus, generally described as Sybil attacks. To prevent this, mechanisms like Proof-of-Work can be used. A proof-of-work is typically a mathematical puzzle that must be solved to perform an action, preventing a node from acting as multiple nodes without the corresponding CPU capacity. A permissionless blockchain becomes an open system that every participant can join and leave at any moment. This allows the recruitment of numerous participants for blockchain maintenance. However, the approach also has some disadvantages. For instance, proofs-of-work are very expensive from the point of view of energy consumption and limit the system throughput. It is also important to point out that this approach allows processes to collude to solve the PoW faster in a byzantine context, and the blockchain becomes less decentralized [4].

2.1.2 Permissioned

In permissioned blockchains, the participants are known to each other and must be authorized by some trusted entity. Since they are more restrictive, it allows running traditional consensus algorithms more easily. Most permissioned blockchains are based on Byzantine

Fault Tolerant State Machine Replication (BFT-SMR) algorithms, which use less energy on processes running a BFT protocol to agree on the transaction order. Compared with permissionless blockchains, it is provably secure and faster performance-wise [4]. They employ a Byzantine failure model with N replicas in which the system can tolerate $\frac{N-1}{3}$ faults and the quorum size is $\frac{N+f+1}{2}$.

2.2 Byzantine Consensus

Byzantine consensus allows correct processes to agree on a single outcome even when some participants exhibit arbitrary behaviour [6].

2.2.1 Nakamoto's Consensus

The consensus algorithm presented by Nakamoto in Bitcoin [3] was the first to be used in permissionless blockchains. Here, the participants (called miners) need to solve the mathematical puzzle (mentioned as Proof-of-work) to participate in the consensus. When a miner solves the puzzle, he is rewarded, which serves as an incentive to participate in the system. In case two processes solve the puzzle simultaneously, it creates a *fork* where there are differences in the local blockchain of certain participants. A *fork* is solved by the *longest chain rule* where, if two or more blocks have been proposed in conflicting branches of the chain, the participants must choose the branch that is believed to have the most work done. There are some disadvantages. The waste of blocks proposed in the conflicting branch, combined with proof-of-work-based membership selection, leads to the waste of mining power and high energy consumption [5].

2.2.2 BFT Consensus

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

A BFT consensus algorithm needs to satisfy the following properties [6]:

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

Definition 2 (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 3 (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.*

Both validity notions are acceptable. Weak validity allows correct processes to decide on the initial value of a Byzantine process. With strong validity, however, this is only possible if not all correct processes have the same initial value. Either way, validity states that all operations accepted need to be proposed by a client. However, it does not differentiate between an honest and a dishonest client, creating the possibility of leader-driven censorship where the leader does not include the honest client's proposed transactions. We will further detail in Section 3.2 the leader's impact on these systems. They are the ones that determine which transactions are going to be satisfied. So the concern of censorship is a fundamental matter that goes beyond the validity property of classic consensus. One solution is to have a rotating-leader approach, hoping a leader will eventually be honest and break censorship. Nevertheless, this could be better since the sequence of leaders is well-known, and the adversary can choose to corrupt the current and next-in-line leaders. Therefore, it becomes essential to have a way to randomize the schedule of leaders to create a moving target for the adversary [7].

Definition 4 (Integrity). *No correct process decides twice.*

Definition 5 (Agreement). *No two correct processes decide differently.*

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

BFT algorithms can be categorized as leader-based or leaderless. In [8], they are categorized in further detail as:

- Efficient leader-based BFT: target performance to achieve high throughput and low latency in fail-free executions. Make sure the normal-case operation is the most efficient possible[9,2,10].
- Robust leader-based BFT: target maintaining availability against potential failures instead of achieving impressive performance in normal-case operations. The main concern is fault tolerance[11].
- Leaderless BFT: tackles the single point of failure on the leader-based approaches and amortizes the coordination of work among a group of or all replicas

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

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

3 Related Work

This section discusses some important features of consensus protocols used to implement distributed ledgers. We only consider leader-based protocols where, for each consensus instance, there is a process that is responsible for proposing a value (in our case, a block). In all these protocols, the execution of a consensus instance requires multiple rounds of communication (sometimes also called phases). The exact number of phases depends on the details of each algorithm. In our analysis, we cover three different aspects of the operation of

these protocols: how processes communicate in each round (described in Section 3.1), how the leader is selected for each instance (detailed in Section 3.2), and whether pipelining is used or not (described in Section 3.3). In Section 3.4 we present several relevant systems and analyze them according to the previous three aspects. In the end, in Section 3.5, we provide a comparative analysis of the previously discussed systems.

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

3.1 Communication Patterns

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

3.1.1 All-to-all

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

3.1.2 Star Topology

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

3.1.3 Tree Topology

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

from its children and forwards it upwards. This process is repeated until the information reaches the root. Subsequently, the leader aggregates the information collected from its children and uses the tree to push the aggregated information to all participants.

3.1.4 Random Topology (Gossip)

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

3.1.5 Effects of the Topology on the Protocol

The topology has an impact on the protocol. When an all-to-all pattern is used, processes communicate directly with each other, and correct processes can exchange information in a single communication step. When a star topology is used, at least two communication steps are required to execute a protocol round. With a tree or gossip topology, even more, steps are required to execute a single protocol round. Furthermore, consensus protocols require multiple protocol rounds to terminate. A protocol such as PBFT [9], based on an all-to-all pattern, uses three communication rounds. HotStuff [2], which uses a star topology, requires four communication rounds. The extra round is needed because, when using a star topology, reliable communication between correct processes in a given round is no longer guaranteed (because the root of the star may be faulty and omit messages). As will be discussed in Section 3.2.2, using pipelining techniques can mitigate the latency increase caused by additional protocol round and communication steps in each round.

It is important to mention why multiple rounds are needed and their goal. We need at least a round to collect votes on a specific value and another round to propose the value to all processes and acknowledge it. The processes, however, can receive several values to vote. In a byzantine setting, there are several reasons for a process to receive conflicting values. It can be a byzantine leader that sends one value to some processes and another to others, or it can be a byzantine process with arbitrary behavior. Either way, there needs to be extra message exchanging to vote from a quorum of votes. Not only do we need multiple rounds of voting when we reach a decision, this one is unrevokable, and we need to inform the decision to all processes. This results in the three rounds for PBFT [9] and four for Hotstuff [2] mentioned.

3.2 Leader Change

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

3.2.1 When to Change the Leader

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

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

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

There are however also some disadvantages to keeping the same leader across multiple instances. In a blockchain setting, the leader selects which transactions are included in a given block, thus the leader has the power to censor requests from clients. This means that a Byzantine or rational leader can drop requests selectively, influencing the fairness of the blockchain for a long time in a stable-leader approach. Also, the stable-leader approach has been shown to be vulnerable to performance attacks [12]. Performance attacks are described as occurring when processes send correct messages slowly but without triggering protocol timeouts. This means that these processes are correct in the eyes of the BFT properties but can degrade performance drastically. This effect can be exacerbated by other mechanisms used to tolerate network instability. For instance, many protocols double the timeout values when the network is unstable but have no clear mechanisms to reduce timeouts again, allowing faulty processes to perform even slower without triggering timeouts.

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

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

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

3.2.2 Leader Change Procedure

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

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

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

3.2.3 Selecting the Next Configuration

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

Definition 6 (Robust Star [1]). *A star is said to be robust if the leader is correct, and non-robust if the leader is faulty.*

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

Definition 7 (Robust Tree [1]). *An edge is said to be safe if the corresponding vertices are both correct processes. A tree is robust iff the leader process is correct and, for every pair of correct processes (p_i) and ((p_j), the path in the tree connecting these processes is composed exclusively of safe edges.*

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

3.3 Pipelining

Consensus protocols require multiple rounds of communication to terminate [13]. In chained consensus, if one starts the $n+1$ instance only after instance n has terminated, the throughput of the system is limited by the latency of the communication rounds. One way of circumventing this limitation is to start optimistically instance $n+1$ before instance n terminates. In chained consensus is possible if the leader of the instance $n+1$ knows the value proposed in instance n. This technique is known as *pipelining*. With pipelining, different rounds of multiple consensus instances are executed in parallel (for instance, the first round of instance $n+1$ is run in parallel with the second round of instance n). Often, messages from multiple instances can be piggybacked in a single network packet. To be effective, the pipelining technique needs to be carefully parameterized according to the system characteristics.

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

In a rotating-leader approach, one common way of pipelining is with a mechanism called Leader-Speaks-Once. Each leader proposes and certifies a single block and rotates so that we limit the leader's influence. Meaning, the leader of the instance n will execute its first round of consensus. Then, a new leader will do the first round of the instance $n+1$ with the second round of the instance n. This is repeated, for all rounds of consensus. Thus, if a protocol requires 4 consensus rounds, 4 leaders will be involved, and each will coordinate a different round. Also, in the same protocol, a single process may be required to execute 4 distinct rounds of 4 distinct instances of consensus in parallel.

However, even though this approach is desirable for blockchains, it introduces liveness concerns [14]. One concern is defined as:

Definition 8 (Consecutive Honest Leader Condition (CHLC) [14]). *Chained Leader-Speaks-Once protocols require the formation of k QCs in consecutive views to commit a block (where $k \in \{2, 3\}$ depending on the protocol).*

CHLC can be a significant exploit for a byzantine process. The concern is transversal to most protocols that combine chaining and a rotating-leader approach. For example, applying this protocol in Hotstuff [2], the leader will coordinate one phase and rotate. The chain structure can be used to pipeline the commands and have one Quorum Certificate(QC) serving as a QC for each of the Hotstuff phases for different blocks as depicted in Figure 1.

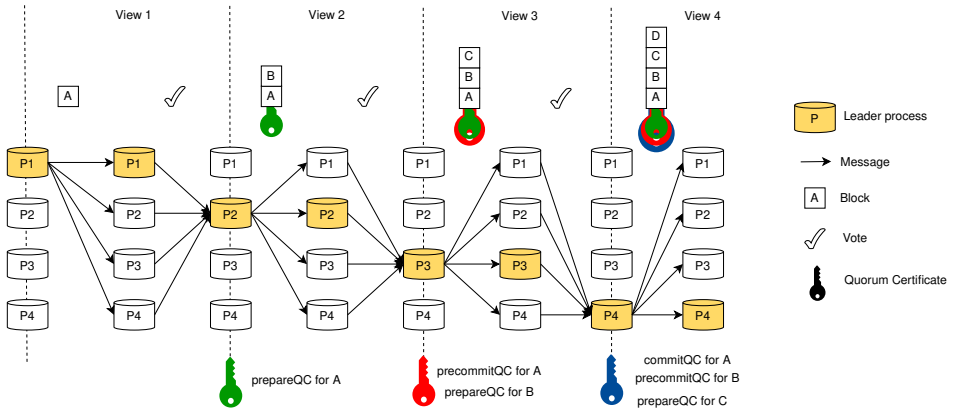

Fig. 1: Chained LSO Hotstuff

Assuming a baseline protocol needs three voting rounds, one for the proposing stage and two for the committing stage. The first phase mentioned as *Prepare phase* creates a *prepareQC* ensuring agreement on block A. In the second and third voting round, a *pre-commitQC* and a *commitQC* are created, respectively, so the processes become locked on a value to decide. Figure 1 demonstrates an example of a chained version of the protocol described above trying to commit four blocks A, B, C, and D. In view four, the QC certifies block A as a *commitQC*, *pre-commitQC* for block B, *prepareQC* for block C and D is proposed. For A (the first block proposed) to be committed, we need three QCs and four consecutive honest leaders. If a failure occurs in view 4, the protocol has to restart the committing process for the block to preserve safety. Thus, for a total number of 4 replicas, a single faulty leader can prevent the protocol from committing any block.

A recent protocol Siesta [14] proves that CHLC cannot be relaxed for arbitrary faults, but proposes a way to relax the condition for the case where we have faults by omission.

3.4 Some Relevant Systems

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

PBFT [9]. This PBFT protocol uses an all-to-all communication pattern and a stable-leader approach. It is a practical algorithm that tolerates faults in an asynchronous Byzantine environment. Previous systems assumed a synchronous system and relied on failure detectors. In PBFT the normal-case operation involves three steps of all-to-all communication. The execution of the protocol is depicted in Figure 2. When clients make requests, the leader proposes an order for the requests by sending a PRE-PREPARE message to the replicas. By receiving this message, the replicas acknowledge the proposal by sending a PREPARE message to each other. After collecting $2f + 1$ matching PREPARE messages, a replica will send a COMMIT message to others. When a replica receives 2f + 1 matching COMMIT messages, she knows that a consensus was reached by sufficient correct replicas

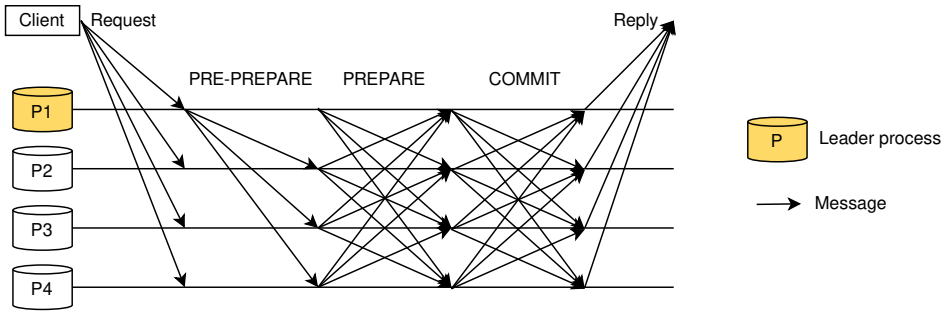

Fig. 2: Normal-case PBFT protocol

and can execute the operations in the transaction and reply to the client. Since PBFT is a leader-based protocol, the replicas have to monitor the leader's state to guarantee the system's liveness when the leader fails. Therefore, every replica starts a timer when a request is received. When this timer expires, processes double its value; if the system fails to make progress with the larger timeout value, the view-change protocol is started. Thus, PBFT follows a stable-leader approach.

PBFT[9] is vulnerable to performance attacks when malicious replicas collude to manipulate the value of the timer. The value of this timer doubles during view-change as part of the protocol, and when it becomes large, a faulty leader can deliberately delay some client requests [4]. The quadratic communication complexity in the normal-case protocol is also a very limiting issue regarding system scalability.

HotStuff [2]. Hotsuff[2] uses a star topology and assumes a rotating-leader approach with a round-robin sequence. It introduces a new goal in BFT algorithms: a system with a linear view change and optimistic responsiveness. Informally, optimistic responsiveness requires that a non-faulty leader, in beneficial circumstances, can drive the protocol to consensus in time depending only on the actual message delays, independent of any known upper bound on message transmission delays. The execution of the protocol is illustrated in Figure 3. The leader broadcasts to the replicas and collects votes from them. This process is repeated

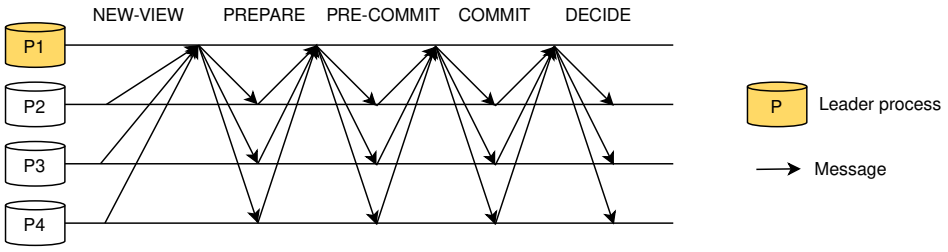

Fig. 3: Normal-case Hotstuff protocol

in three phases. This one-to-all communication pattern allows the reduction of the message complexity from $O(N^2)$ to $O(N)$. However, compared to PBFT, the number of steps is increased from three to seven. In PBFT, we have a communication step per phase, and

now we have two steps per phase since the leader needs to disseminate and aggregate the replicas' votes. Hotstuff uses pipelining techniques to compensate for the extra steps and improve throughput. As a result, it can execute four instances in parallel, thus improving the throughput by four [15]. However, the leader needs to disseminate N messages per round and receive a quorum of votes: this creates a bottleneck resulting in limitations to scalability [1].

Kauri [1]. Kauri is a consensus protocol that uses a tree topology to perform vote aggregation and disseminate information to address scalability limitations like the leader bottleneck mentioned in Hotstuff [2]. When in a tree topology, each process has to disseminate at most m messages and processes at most m votes, where m is the number of children a node has in a tree (also referenced as the fanout). Thus, the process is busy for a shorter time and has longer idle times.

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

Kauri follows a stable-leader approach, and the view-change protocol can be seen as a one-to-all handoff even though we have a tree topology. The internal nodes aggregate the children's votes into one single vote, which is O(m), the verification process is O(1), and the worst-case scenario is linear complexity. When there is a faulty leader, Kauri follows the same strategy as Hotstuff, where the replica starts a view-change, the new tree configuration is built, and sends a NEW-VIEW message directly to the next leader. The new leader receives the messages and either continues the previous work or proposes a new one. It still does not surpass linear complexity. The communication pattern is depicted in Figure 4

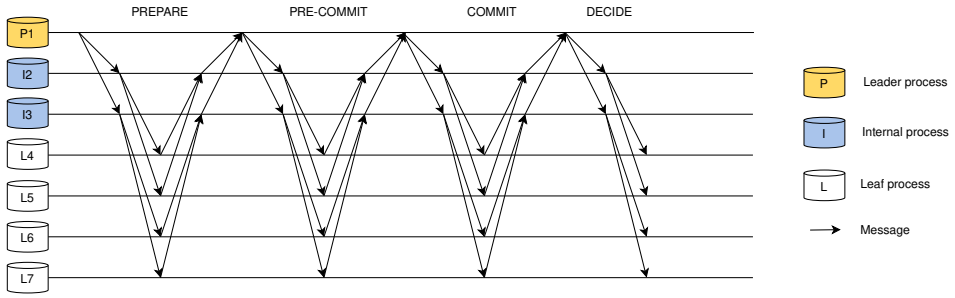

Fig. 4: Kauri's communication protocol

The main difficulties in tree topologies are the additional latency and the complex reconfiguration strategy. While trees allow a more balanced way to distribute the load among replicas, there is an additional round-trip in the dissemination and aggregation patterns affecting system throughput [1]. To mitigate the additional latency, Kauri presents a novel pipelining technique. The pipeline stretch is a parameter that tells the system how many instances of consensus can send to use the extra time he has because of the tree topology. This technique is depicted in Figure 5. This pipeline stretch is based on the following:

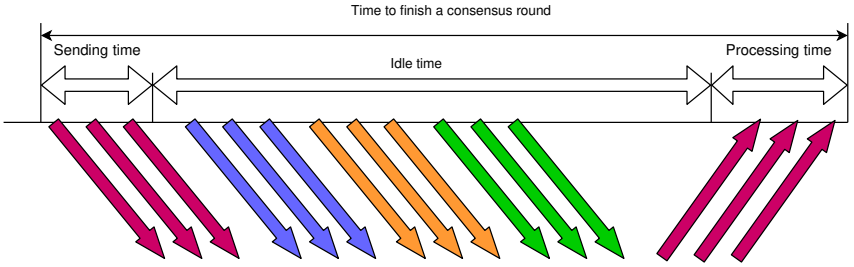

Fig. 5: Representation for pipeline stretch of four in a consensus round.

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

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

In Kauri [1] is represented as $\frac{mB}{b}$.

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

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

Byzcoin [7]. Byzcoin is a solution that, like Kauri [1], leverages a tree topology but employs a rotating-leader approach. It uses a tree communication pattern to optimize transaction commitment and verification during normal-case operations while guaranteeing safety and liveness. Trees are built randomly from a VRF, creating a non-deterministic sequence of configurations. Byzcoin wants to apply trees in Bitcoin [3], which is not a permissioned blockchain. To address this challenge, it creates small groups dynamically from windows of recently mined blocks and performs consensus in each. Furthermore, to maintain the fairness-enforcing benefit of Bitcoin's leader election, Byzcoin makes a PBFT view-change every time the leader signs a proposal. If there is a conflict, another view-change is performed in which the successful miners attempt to persuade other participants to adopt their block. Finally, when a miner creates a block, it forms a tree for collecting signatures with himself as the root. Like many BFT protocols in practice, this protocol is still vulnerable to slowdown or temporary DoS attacks by byzantine nodes, excluding processes from consensus and censoring transactions.

Prosecutor [10]. Prosecutor has a star topology and follows a stable-leader approach. It intends to penalize Byzantine servers, ensuring they do not usurp leadership over time. When we have a faulty leader, no consensus can be processed, and the system becomes unavailable. This is a problem for all leader-based BFT consensus algorithms. The Prosecutor's dynamic-penalization election technique mitigates this problem by reducing the time the system is unavailable. The star topology allows the achievement of linear message complexity. It creates a novel technique to penalize Byzantine servers allowing replicas to actively claim to

be the new primary and apply computation penalties (PoW) on candidates to primary. The more failures a candidate has exhibited, the greater the computation penalty. This way, it entices replicas to operate correctly to avoid performing computation work.

However, PoW-like penalization may become less efficient if faulty servers have a strong computation capability. In this case, Prosecutor may suffer a long period without a correct leader before Byzantine servers exhaust their computation capability [8].

Spin [11]. Spin [11] has an all-to-all communication pattern and follows a rotating-leader approach. The spinning algorithm tries to deal with PBFT stable leadership vulnerability to performance attacks previously mentioned [8].

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

Nevertheless, when a failure is in a replica and not in a leader, it still incurs extra message-passing, impacting the system performance [8].

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

Carousel [16]. Carousel focuses on the main drawback of the round-robin approach followed in Hotstuff [2]: it is not bounded to the number of faulty replicas that become leaders impacting latency negatively even in crash-only executions. Therefore, it proposes a new leader-rotation mechanism applied in a Hotstuff-based system. It is considered the number of faulty leaders in crash-only executions after the global stabilization time (GST), a property called Leader-utilization. The main idea is to track the involved parties via the records of their participation (for example, signatures) in the chain and elect leaders among them. In other words, allow the use of the reputation of a process to avoid crashed leaders in crash-only executions. At the beginning of a round, each honest party checks if there is a committed block B at round *r - 1*. From this, we can guarantee that the endorsers of B, at least in round *r - 1*, did not crash, adding them to the set of possible leaders. After that, we exclude the f latest authors of committed blocks from the set for chain-quality purposes. Then, finally, the leader is chosen from the remaining. This paper shows that latency and throughput improved in the presence of crash faults compared to the Hotstuff approach with a round-robin election mechanism.

Gosig [17]. Most protocols mentioned above allow adversaries to launch DDoS attacks on honest processes and partition them from others. Moreover, with the leader as a bottleneck, the system can become limited by the slowest process in the propagation process due to participants' different capacities.

In Gosig [17], the consensus protocol operates in rounds and appends one block to the blockchain per round. Each round has a proposing phase. In this phase, the proposers are selected, and each creates a block and broadcasts it to all processes. Next is the signature collection stage, where each process chooses a block he has received to vote on, signs his decision, and relays the aggregated signatures. Under this consensus protocol, a gossip network

is used to propagate all messages. Furthermore, it uses a rotating-leader approach since, for each new block, a random proposer is elected using a verifiable random function(VRF). The protocol is depicted in Figure 6 At the start of each round, some processes secretly become

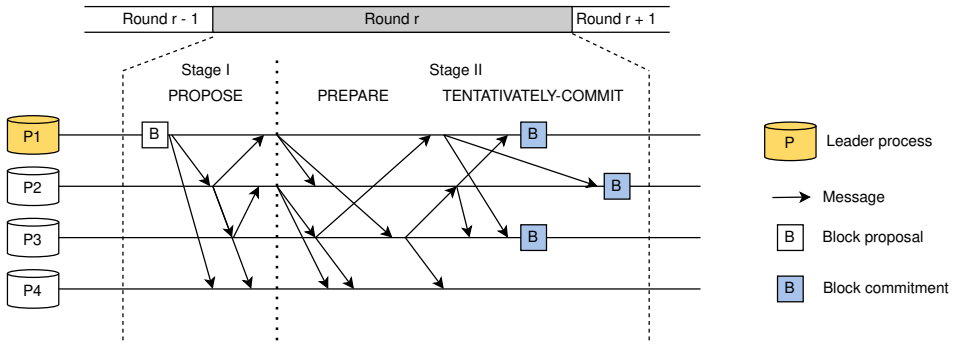

Fig. 6: A round of consensus with Gosig's protocol.

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

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

Algorand [18]. Reaching a consensus in an open setting is hard because anyone can participate, and an attacker can create several pseudonyms (Sybils), making the traditional consensus impossible to apply. Other protocols use proof-of-work like Bitcoin [3], but Algorand uses a Byzantine agreement protocol. Algorand wants to avoid Sybil attacks, scale to millions of users, and be resilient to DoS attacks. To address these goals, he prevents Sybil attacks by assigning weights to each participant and guaranteeing some fraction of the weight is from honest users. In addition, Algorand achieves scalability by choosing a committee to run each step of the protocol. To prevent an attacker from targeting committee members, he selects the committee in a non-interactive way by using VRF; the attacker will only know which participant to target once he is already participating in the consensus. This protocol is implemented over a gossip network, and the committee members can be replaced after every step to mitigate attacks on them following a rotating-leader approach. However, a committee solution has the drawback of affecting the system's resilience.

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

There have been several attempts to reduce the number of phases Hotstuff [2] needs to commit. However, these approaches always have a trade-off: a more expensive view-change for a two-phase normal case. Marlin [19] is an example that achieves consensus in two phases in the normal case and has at most three phases in view-change executions. Zyzzyva [20] introduces a linear path into PBFT. Thus, this optimistic path has linear complexity, while the leader replacement protocol remains $O(N^3)$.

On the other hand, the authors of Aardvark [21] discuss that a limited normal-case performance reduction is preferable to the existing loss in availability on the view-change execution.

Aardvark [21] uses monitoring solutions to decide when to change leaders. It leverages PBFT view-change protocol, but the correct primary does not dominate forever. Instead, it imposes two expectations: the primary has to be sufficiently and timely issuing PRE-PREPARE messages; maintain sustained throughput. The replicas monitor the leader; they initiate a view change if he fails these expectations. It still leaves uncertainty since correct leaders can be mistakenly replaced with the system making no process and imposing unnecessary view-changes [8]. Still, the authors reinforce that the benefits of evicting a faulty leader outweigh the recurring costs of performing view-change regularly.

3.5 Discussion

Table 1 provides a comparison of systems covered in Section 3.4. In our analysis, we focus on seven key factors. The first two factors capture the communication pattern used to execute rounds and the strategy used to change leader. Next, for the *Non-deterministic sequence* factor, we focus on whether systems use a deterministic sequence of leaders; systems that use a deterministic sequence of leaders may be more vulnerable to attacks, given that attackers can exploit this knowledge to their benefit. The *Leverages pipelining* factor captures if the systems use pipelining techniques to improve throughput. The two following factors are regarding load balancing. We consider two definitions of load balancing in this discussion: intra-round load balancing (the load is distributed within a round) and cross-round load balancing (the load is distributed in the overall execution across multiple rounds). Regarding load balancing on nodes, Kauri [1] offers intra-round load balancing by distributing the load in a tree topology. On the other hand, various systems like PBFT [9] and Spin [11] provide cross-round load balancing. Here, the leader's load is distributed per all processes within several rounds. Therefore, in the *Load balancing on nodes* factor, we focus on the cross-round definition, whether the system is able to distribute the computational and bandwidth usage among different nodes across rounds. In terms of load balancing on links, we follow the same logic. In factor *Load balancing on links*, we analyze whether the system can distribute the load among multiple links across rounds. A fair distribution of the link load can avoid persistent bottlenecks. The last factor indicates whether knowledge regarding the network conditions is used to achieve better performance.

We described two systems with a clique topology: PBFT [9] and Spin [11]. PBFT still has some issues that can be improved, like the number of resources needed, the vulnerability to performance attacks mentioned in Section 3.2.1, and the significant message complexity of the protocol. Spin tries to solve the vulnerability to performance attacks. It uses a rotating strategy with a blacklisting technique for the processes suspected to be faulty. This technique allows for a rotating approach but without a faulty process periodically becoming the leader. Other solutions like Aardvark [21] also solves problems related to attacks on performance. However, its solution to prevent faulty leaders from delaying the service is less efficient than

	Topology	Leader election approach	Non-deterministic sequence	Leverages pipelining	Load balancing on nodes	Load balancing on links	Aware of network conditions
PBFT [9]	Clique	stable	✗	✗	✗	✗	✗
Spin [11]	Clique	rotating	✗	✓	✓	✗	✗
Prosecutor [10]	Star	stable	✗	✗	✗	✗	✗
Hotstuff [2]	Star	rotating	✗	✓	✓	✓	✗
Carousel [16]	Star	rotating	✗	✗	✗	✓	✗
Gosig [17]	Random	rotating	✓	✓	✗	✗	✗
Algorand [18]	Random	rotating	✓	✗	✓	✗	✗
Kauri [1]	Tree	stable	✗	✓	✗	✗	✗
Byzcoin [22]	Tree	rotating	✓	✗	✓	✓	✗
Kauri Adaptive	**Tree**	**rotating**	✓	✓	✓	✓	✓

Table 1: Comparison of existing algorithms.

Spinning. Aardvark also changes the leader when it suspects faulty behavior by running a view change operation. However, Spinning does not incur the cost of running a distributed algorithm with several communication steps. As a result, Spin becomes more efficient than Aardvark. In addition, it allows for load balancing among the processes across rounds.

For star topology, we described three more systems: Hotstuff [2], Prosecutor [10], and Carousel [16]. Hotstuff primary advantage is simplicity, enabling pipelining techniques and building large-scale replication services. Some disadvantages are the bottleneck on the leader and the leader election being performed by following a round-robin sequence. A faulty process will periodically be the leader. Also, this round-robin sequence is deterministic and predictable, making the system vulnerable to attacks on the leader. Prosecutor focuses on the impact of having a faulty leader periodically and uses a dynamic-penalization election technique to mitigate this problem, reducing the time the system is unavailable. However, it imposes computational costs on faulty servers, which is a different approach from others since the previous approaches on penalization allow faulty servers to be freely released after pretending to be correct servers. On the other hand, PoW-like penalization may become less efficient if faulty servers have a strong computation capability. Carousel also presents a new technique to deal with the impact of the round-robin approach. It focuses on minimizing the effect of crash-only executions by a participation tracking technique. This new leader-rotation mechanism demonstrated drastic performance improvements in throughput and latency compared with Hotstuff-based systems with a round-robin mechanism.

In the category of Random topology, we mentioned two systems: Gosig [17] and Algorand [18]. These two systems provide solutions to the vulnerability of attacks on the leader. Gosig focuses on minimizing attacks on the leader using a verifiable random function (VRF) so attackers do not know the sequence. Using this method means that the leader is only known when the consensus round starts, so the system tolerates attacks on the leader. Algorand selects consensus groups, using a VRF to prevent attacks targeting the leaders. However, this system has the drawback of using a committee approach for consensus, which limits the system's resilience.

Lastly, categorized as a tree topology, we analyzed two existing systems: Byzcoin [22] and Kauri [1]. Both systems use a tree communication pattern to distribute the computational and bandwidth load among processes intra-round. Byzcoin builds trees, using a VRF, to collect signatures when a miner creates a block. However, the throughput depends on the round latency and does not provide a quick recovery. The main advantages of Kauri are load

balancing for scalability and a quick recovery strategy. In addition, using a novel pipelining technique, Kauri achieves high throughput as the system grows. However, Kauri uses a stable-leader approach creating fairness concerns.

We propose Kauri Adaptive, a system with Kauri as the baseline achieving intra-round load balancing on the nodes and scalability while avoiding throughput limitations due to additional round latency of the tree topology. In addition, we will introduce a rotating-leader approach in Kauri to guarantee fairness, a new non-deterministic sequence approach for the tree configurations to limit attacks on the leader, and a new technique to achieve load balancing on the links and nodes across rounds. Table 1 shows that for every topology, there is at least a system that follows a rotating-leader approach and leverages a pipelining technique. There is no version of a system with a tree topology, following a rotating-leader approach, that leverages pipeline. Thus we propose a new rotating-leader approach that will leverage Kauri's pipelining technique. Finally, the proposed system will leverage network conditions to achieve better performance.

4 Architecture

In this section, we detail our proposal. Our goal is to implement a rotating-leader strategy in Kauri to guarantee load balancing while maintaining the good performance results of the system. We intend for our new tree rotation algorithm to allow each participant to be a leader only once while considering the loads in links and nodes and pipelining efficiency. This rotating-leader approach results in fairness and censorship resistance in a blockchain.

4.1 Challenges

Implementing a rotating-leader strategy means that the view-change/leader election will be in the common path of the executions. Regularly reconfiguring the tree comes with certain disadvantages that we want to resolve. First, it is more likely to find non-robust trees; thus, there are more periods where the systems' throughput is negatively affected. Second, it is harder to pipeline effectively, as the leader is only in this position for a limited time, negatively affecting throughput.

Regarding reconfiguration strategies, there are two main challenges. First, we must decide the tree shape, in particular the values of height and fanout in a tree. Both these factors impact the dissemination process in a tree. For example, we will have a higher round-trip latency if we have a tree with a significant height factor. On the other hand, if we have a considerable fanout factor, we will impact the time needed for dissemination and limit the potential for pipelining. Second, we need to choose the position and role for each process to be in the tree, and there are factorial possibilities.

We defined the following goals to follow when defining our solution:

1. Allowing every participant to be a leader and coordinate consensus.
2. Make sure the improvement made by the pipelining technique is not wasted.
3. Achieve load-balancing in the tree (nodes and links).
4. Leverage network heterogeneity.

For the first goal, we intend to devise a strategy that allows each participant to serve as a leader at least once. For the second goal, rotating the leader frequently raises concerns about how to maintain the pipeline started by the previous leader. For example, if the next

leader is a leaf node, receiving the proposals from the previous leader, takes longer before proposing their block. Therefore, we want an approach that ensures the pipeline is not wasted, allowing for a good system throughput in its tree topology. The third goal can be analyzed from two angles: the nodes' and links' loads. In terms of load on a node, a node's role in a tree configuration is related to its load. For example, an internal node has more work than a leaf, so we want to guarantee that all nodes in the system evenly distribute this work. Regarding the load on links (parent-child), we want to achieve a balanced solution not to saturate any links and exploit link diversity. There are already systems that measure latency between nodes in a Byzantine system [23]. However, even though this is already solved, given a set of latencies between nodes, we still need to decide where those nodes go in a tree topology. Therefore, we want to ensure that our strategy considers this for our fourth goal. Furthermore, to reconcile the third and fourth goals, we want to guarantee that the load balance is proportional to the node capacity.

4.2 Preliminary solution

We designed a preliminary algorithm that solves some of the requisites mentioned in Section 4.1, but not all. Next, we will explain the intuition behind the algorithm, the problems it solves, and those left open. We divided our approach into two distinct problems: i) generate all trees (described as the *Generation phase*); ii) given the set of generated trees, define their sequence (described as the *Sequence phase*).

In the *Generation phase*, our goal is to generate N trees (with N as the total number of processes) that meet the following requirements: have different roots; every node has the same load after N generations; every link has the same load after N generations. Our starting point for the generation strategy is based on thinking of a tree as a list with the root as the first element and then appending the nodes left to right per level. This is depicted in Figure 7. We generate a tree with N nodes and generate N trees from her by applying a

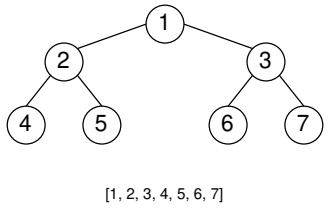

Fig. 7: Tree structure for a system with seven nodes and corresponding list representation

shifting technique of one step to the left for each generation as described in Figure 8. This preliminary solution allows us to achieve different roots for every tree and a distributed load on the nodes. In Section 4.3, we will further discuss what is still missing in this generation strategy.

In the *Sequence phase*, we focus on creating the sequence of the generated trees. Our goal is to create a strategy that meets the following requirements: guarantees that the leader in configuration i was an internal node in the configuration i - *1*; the sequence is not predictable for an attacker to exploit the system. Our preliminary solution creates the sequence by adding each configuration sequentially generated by the list-shifting techniques.

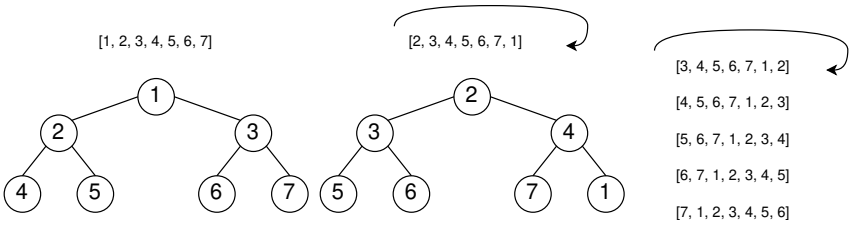

Fig. 8: Generation of trees in a seven-node system applying the shifting technique

This solution allows a child of the previous root to be the next leader but is a predictable sequence for an attacker. We discuss the limitations of the current approach next.

4.3 Discussion

Regarding the first goal mentioned, the shifting technique presented will have a different root for N generated trees (N being the total number of processes), and all participants will be a root only once, fulfilling the goal.

For the second goal, to maintain the pipelining improvement on throughput, it is beneficial that the leader of the new configuration has been a child of the root in the previous configuration. This way, the new leader receives the proposals of the prior leader faster. A list-shifting technique to the left seems most appropriate for this. Thus, the leader's leftmost child in the current configuration will be the leader in the following configuration.

We can also analyze this preliminary solution regarding load-balancing for the third goal. In terms of the load difference in a leaf and an internal node, this approach allows for every node to be internal the same amount of times. Thus, in the overall execution of the protocol (cross-round), the extra load on internal nodes will be evenly balanced between all nodes. Regarding the load on links (parent-child), this approach does not allow each link to be used only once to achieve a uniform distribution. For example, in Figure 9, in the process of generation, we can see that in the second tree, there are common links between the previous (dashed links) and the next generated tree (red links). Regarding the load-balancing of the links, we have links that are only used once and others that are used more than once. We intend to look into this further and find a solution that considers this balance.

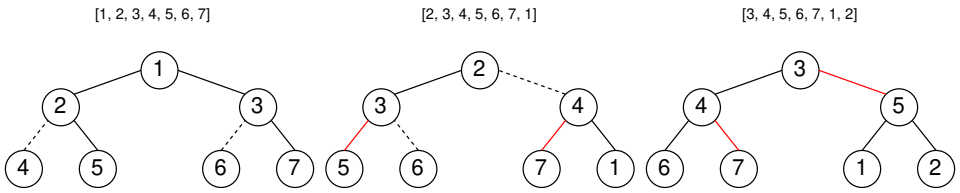

Fig. 9: Generation of next tree in a seven node system

However, due to the heterogeneity of where these systems are deployed, an approach in which some links have more load should not be discarded. We plan to investigate further how, even with some links carrying more load than others in a heterogeneous environment, we could attribute these links to the strongest nodes or links with more capacity.

5 Evaluation

In this section, we describe our evaluation plan. Our main goal is to design and implement a strategy to rotate the leader in the Kauri protocol to avoid the limitations of the stable leader approach and promote better load balancing. While doing so, we will attempt to minimize the reconfiguration's negative effects on the Kauri pipelining. Therefore, we would like to maintain or even improve the performance of Kauri's current version (based on a stable leader). In detail, we want to answer the following questions: i) what is the impact of a rotating-leader strategy on the system throughput compared to Kauri; ii) how the latency of the proposed system develops for different block sizes and bandwidth compared to Kauri; iii) what is the impact on the system performance to distribute the load on the nodes cross-round; iv) what is the impact on the system performance to distribute the load on the links cross-round; v) is the sequence of leaders known to an attacker; vi) how the proposed solution behaves in a heterogeneous deployment.

For comparison, we will use the current stable implementation of Kauri and a similar workload to the one used in Kauri. We plan to start with configurations that use a tree of depth 2, and if time allows, we will also experiment with deeper trees. In the following, we present our evaluation metrics.

Latency We will conduct a study concerning the latency of our approach compared with Kauri. To evaluate latency, we measure the time it takes for a single quorum to be obtained for a given set of transactions (block).

Throughput We plan to evaluate the throughput of our approach compared with Kauri. We want to measure the impact in our system of the pipelining technique from Kauri. We will fixate the bandwidth, block size, and the number of processes in the system and vary the round trip latency (and by the performance model, the pipelining stretch by consequence). Kauri shows a stable throughput independent of the inherent system latency. Thus, the authors conclude that it can fully leverage the available computational and networking resources independent of the system latency. We will measure the throughput when we do the same experience for the proposed solution.

Load balance on nodes We plan to evaluate our systems in terms of load-balancing compared to Kauri. Considering the tree structure and each node's role, we will analyze the amount of load each process has across rounds. We will analyze how often a node is a leader, an internal node, and a leaf across rounds.

Load balance on links We will also evaluate how much load each tree link has in the overall execution of the protocol.

Non-deterministic sequence of leaders We plan to evaluate whether the sequence of leaders is previously known to attackers.

6 Schedule of Future Work

Future work is scheduled as follows:

- Mid-January - May: Detailed design and implementation of the proposed architecture, including preliminary tests.
- May: Perform the complete experimental evaluation.
- June: Write a paper describing the project.
- July - Mid-September: Write the dissertation.
- Mid-September: Deliver the MSc dissertation.

7 Conclusion

Most permissioned blockchains are based on variants of Byzantine fault-tolerance (BFT) consensus protocols that have the problem of scalability since all participants engage in multiple rounds of data exchange. Kauri [1] combines dissemination/aggregation trees and pipelining to distribute the load and compensate for the increased latency of using trees. However, it uses a stable-leader approach to leverage the benefits of pipelining. Several arguments favor changing the leader between consecutive consensus rounds to provide censorship resistance. These arguments motivate the need for a strategy to rotate the leader with a minimal negative impact on pipelining and promoting a good load balance among all participants.

In this report, we surveyed the state-of-the-art BFT algorithms. We analyzed their communication patterns and discussed leader rotation strategies and the impact of the pipelining techniques. We elaborated a solution to support a rotating-leader approach in a tree topology while maintaining an acceptable throughput in this environment. Finally, we defined the evaluation methods and presented the scheduling for future work.

References

1. R. Neiheiser, M. Matos, and L. Rodrigues, "Kauri: Scalable BFT consensus with pipelined tree-based dissemination and aggregation," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 35–48.
2. M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus in the lens of blockchain," *arXiv preprint arXiv:1803.05069*, 2018.
3. S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.
4. X. Wang, S. Duan, J. Clavin, and H. Zhang, "BFT in blockchains: From protocols to use cases," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–37, 2022.
5. D. Yaga, P. Mell, N. Roby, and K. Scarfone, "Blockchain technology overview," *arXiv preprint arXiv:1906.11078*, 2019.
6. C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.
7. E. Kokoris-Kogias, "Robust and scalable consensus for sharded distributed ledgers," *Cryptology ePrint Archive*, 2019.
8. G. Zhang, F. Pan, M. Dang'ana, Y. Mao, S. Motepalli, S. Zhang, and H.-A. Jacobsen, "Reaching consensus in the byzantine empire: A comprehensive review of bft consensus algorithms," *arXiv preprint arXiv:2204.03181*, 2022.

9. M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.
10. G. Zhang and H.-A. Jacobsen, "Prosecutor: An efficient bft consensus algorithm with behavior-aware penalization against byzantine attacks," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 52–63.
11. A. N. Bessani, G. Veronese, L. Lung, and M. Correia, "Spin one's wheels? byzantine fault tolerance with a spinning primary," 2009.
12. Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 197–206.
13. T. H. Chan, R. Pass, and E. Shi, "Pala: A simple partially synchronous blockchain," *Cryptology ePrint Archive*, 2018.
14. I. Abraham, N. Crooks, N. Giridharan, H. Howard, and F. Suri-Payer, "It's not easy to relax: liveness in chained bft protocols," *arXiv preprint arXiv:2205.11652*, 2022.
15. S. Alqahtani and M. Demirbas, "Bottlenecks in blockchain consensus protocols," in *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*. IEEE, 2021, pp. 1–8.
16. S. Cohen, R. Gelashvili, L. K. Kogias, Z. Li, D. Malkhi, A. Sonnino, and A. Spiegelman, "Be aware of your leaders," in *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*. Springer, 2022, pp. 279–295.
17. P. Li, G. Wang, X. Chen, F. Long, and W. Xu, "Gosig: a scalable and high-performance byzantine consensus for consortium blockchains," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 223–237.
18. Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 51–68.
19. X. Sui, S. Duan, and H. Zhang, "Marlin: Two-phase bft with linearity," *Cryptology ePrint Archive*, 2022.
20. R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 45–58.
21. A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults." in *NSDI*, vol. 9, 2009, pp. 153–168.
22. E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing," in *25th usenix security symposium (usenix security 16)*, 2016, pp. 279–296.
23. J. Seibert, S. Becker, C. Nita-Rotaru, and R. State, "Newton: Securing virtual coordinates by enforcing physical laws," *IEEE/ACM Transactions on Networking*, vol. 22, no. 3, pp. 798–811, 2014.