Dynamic Trees for Byzantine Consensus Protocols

(extended abstract of the MSc dissertation)

TOMÁS PEREIRA, Instituto Superior Técnico, Universidade de Lisboa, Portugal
SUPERVISORS: PROFESSOR LUÍS RODRIGUES and PROFESSOR MIGUEL MATOS, Instituto Superior Técnico, Universidade de Lisboa, Portugal

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

Additional Key Words and Phrases: Distributed Systems, Blockchain, Byzantine Fault Tolerance, Consensus

1 Introduction

Byzantine Fault Tolerance (BFT) consensus protocols allow correct processes to reach agreement even in the presence of a fraction of malicious processes. BFT protocols have been first introduced for synchronous systems [1] but have been subsequently extended to execute in eventually synchronous settings [2–4]. BFT protocols require multiple rounds of message exchange among participants and are, therefore, costly, both in terms of communication and processing. For this reason, early implementations considered a relatively small number of participants (in the order of dozens) [5]. However, with the emergence of blockchains and the increasing relevance of large-scale systems based on blockchains, the need to design Byzantine consensus protocols that can scale to hundreds of participants has become a relevant topic. [6]

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

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

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

trees. We achieve this by proposing novel techniques to dynamically reconfigure dissemination/aggregation trees in BFT protocols, namely, a technique that avoids a significant throughput degradation during the reconfiguration of the dissemination/aggregation tree. Additionally, we evaluate and analyse an implementation of our approach in emulated network environments, considering both homogeneous and heterogeneous settings.

2 Related Work

In this section, we analyse relevant features frequently utilized in the design and implementation of state-of-the-art permissioned blockchain BFT protocols.

2.1 Coordinated Agreement

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 protocol designs need to take into account mechanisms that ensure coordination between correct nodes for consecutive consensus instances. We define the following approaches for coordinated agreement:

Leader-based [2–4]. In leader-based BFT consensus algorithms, the role of the coordinator, or as it is usually called, the leader, is to help all correct processes converge towards a decision in a relatively fast manner [11]. In the context of blockchain, it is the leader's role to propose which transactions will be appended to the ledger. Leader-based algorithms may adopt a *stable leader* policy, where reconfiguration occurs only when a significant number of nodes detect either a lack of progress or suspicious behaviour, triggering a forced leader change, or a *rotating leader* policy, where the system proactively rotates its leader to provide the protocol with censorship resistance and workload distribution.

Leaderless [12, 13]. Leaderless coordination aims to mitigate leader-based system issues such as censorship and leader-related bottlenecks through the use of higher decentralization [11, 14]. 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.

Multi-leader [15, 16]. 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. An example is Mir-BFT [15], where there is a throughput gain by allowing a set of leaders to propose independently and in parallel. As a drawback, the system is limited by its high communication costs and complex design requirements [17].

Group-based [7, 18]. In group-based coordination, the system is divided into groups based on factors like responsibilities or node characteristics (e.g., location or latency). A typical design involves a *global* group managing *local* groups, where global nodes collect and share votes from local groups. Consensus is achieved via communication within the global group, and the results are then propagated down to the local groups. GeoBFT [18] is a group-based protocol that optimistically allows each cluster to make decisions independently and only afterwards relays their decisions through a global channel. Its robustness is limited by its high decentralization, however, as every local cluster must verify Byzantine node resilience locally and not at a global scale.

2.2 Pipelining

To compensate for the throughput losses experienced by systems with several communication steps in their consensus execution, protocols can leverage the added latency to their advantage through a concept known as pipelining. Pipelining is based on the idea of optimistically initializing future consensus instances while the current one still has not terminated. This can be done due to the fact that each instance of consensus can be divided into several phases (or rounds) of communication that have idle time between them, as nodes have to wait for the propagation and processing of messages in order to receive the replies necessary to advance to the next round. To exemplify, in systems such as Chained HotStuff [3], the leader can optimistically initiate consensus for block $(n + 1)$ after it receives the proposal from the previous leader for block n, meaning that it simultaneously executes round 1 of communication for block $(n + 1)$ as it executes round 2 of communication for block n. Meanwhile, in Kauri [4], pipelining is further extended by defining the notion of *pipeline stretch*, which exploits the piggybacking functionality of network packets introduced in Chained HotStuff to allow multiple new instances of consensus to be started simultaneously in a singular round of communication of a given consensus instance.

3 Dynamic Reconfiguration

In this section, we propose the techniques to support what we call dynamic reconfiguration in systems such as Kauri. We aim to complement Kauri with a rotating leader policy such that the costs of reconfiguration (more specifically, the impact on the throughput of the pipelining execution) are reduced. Additionally, the tree construction can be encapsulated in a schedule format to be used by the leader rotation. As it will be discussed in our evaluation, the performance of our reconfiguration mechanism can be dependent on various factors such as tree height and fanout, the displacement of the nodes between two different consecutive trees (switching two nodes in the leaf layer might have a different effect compared to switching the root node for a leaf node), the frequency of reconfiguration, and so forth. It is also important to emphasize that the overall performance of the system can be influenced by a smart rotation of trees in heterogeneous system contexts, and for that reason, we also provide an analysis of the benefits of our solution in such environments.

3.1 Model and Assumptions

We consider a protocol that implements blockchain services in a permissioned setting, meaning that the group of participants is known amongst themselves. Given this environment, we assume that a tree schedule is provided to all the participants (i.e., a schedule containing the order of the various configurations the system will rotate through), where each tree has its pipeline stretch associated

with it (i.e., the number of consensus instances that can be initialized optimistically whilst the current one has not been decided).

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

When a consensus instance is initialized by a leader, it uses a given tree to disseminate a block containing client transactions. We denote the tree associated with the said instance as the *initial configuration*. We assume that, in the absence of faults, the initial configuration for all consensus instances is pre-defined. This means that we assume a pre-defined and globally known schedule from which participants obtain the different trees that will be used to execute the first round of communication for each consensus instance.

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

Lastly, the protocol assumes the same behaviour as Kauri when handling Byzantine faults. This means that the dynamic reconfiguration does not affect the original protocol's recovery mechanisms, even during periods of asynchrony or when malicious nodes try to delay the transition between configurations. In the normal case, reconfiguration is triggered in a participant when the last block of a configuration is delivered. In the case of a Byzantine fault during operations, participants will fail to make progress before delivering the current configuration's last block. In this scenario, participants will move on to the next configuration in the rotation and attempt to make progress under the new tree. The duration of configurations is always fixed, meaning that planned rotations will adapt in the case of recovery so that configurations do the expected amount of blocks.

3.2 Schedules

Our rotating leader policy takes a tree schedule as the input that determines the sequence of the different configurations that will be utilized during protocol execution. Trees belonging to a schedule are identified from 0 to n, with n being an arbitrary number. Every participant in the protocol can infer from any tree in the schedule the following configuration details: the root of the tree (i.e., the leader), its own parent (if it has one) and its own children (if they exist). Additionally, each tree in the schedule can have a distinct value for its fanout (i.e., the number of children each internal node has) and for its pipeline stretch, making these values dynamic throughout protocol execution. Lastly, each tree has a *target duration*, defined by the number of blocks during which the tree will remain in effect before the system reconfigures again, assuming the absence of faults. When the system reaches the target block of T_n, the system will reconfigure and resume execution with T_0, making it so that schedules are cyclical. In the current version, in the case of faults, the trees in the schedule are not recomputed.

3.2.1 Advantages of Using Schedules. Given the context of the problem, a rotating leader policy offers additional flexibility in systems that rely on tree topologies, since by allowing the system to choose the structure of the tree, it is selecting which pairs of nodes will establish an edge for communication. Compared to a star topology, where every node is forced to communicate directly with the node that is the leader, in a tree topology we can more easily select the structure that will lead to the usage of communication channels with a better performance in order to drive a higher throughput in the system. However, as the number of possible trees is exponential, exercising a judicious choice of which trees to use is a complex and difficult task, albeit crucial. Considering that the protocol is to be applied in a permissioned blockchain context, we can assume that nodes can obtain an estimate of the quality of the connections between themselves. Thus, a tree schedule can utilize this information to define trees that can obtain an expected better performance, when compared to randomly chosen trees in a geo-distributed WAN. Additionally, we can further improve the throughput of the system by adapting the pipeline stretch of each tree accordingly and by ordering the trees in the schedule in a way that reduces the impact of the reconfiguration on the execution of the pipelining techniques.

3.2.2 A Simple Schedule. As a base for our rotating leader policy, in order to guarantee that a schedule lets every node perform the role of leader (that is, be the root of a tree), we assume that, unless explicitly stated otherwise, executions will utilize a schedule based on the rotation of the tree's participant array. What this means is that each tree in the schedule is obtained by rotating the nodes in the previous tree. This also means that the schedule will have N trees, with N being the total amount of participants in the system. An example of this type of schedule can be seen on Figure 1 for a system with $N = 7$ nodes.

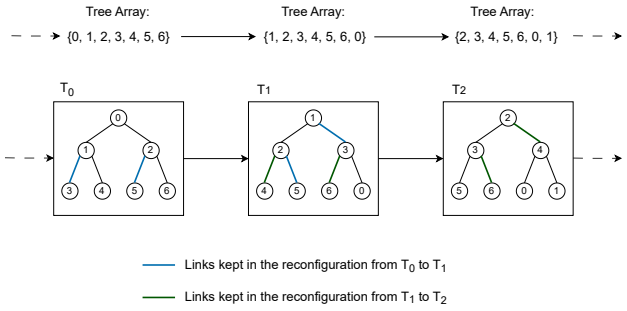

Fig. 1. A simple rotation-based schedule.

This type of schedule makes it so that each node has an equal opportunity to become and act as the leader of consensus. Each

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

3.3 Transitioning Between Configurations

One of the biggest challenges in our work is reducing reconfiguration costs. These costs come from various sources, such as the latency caused by the tree's height, the overhead from switching connections and updating the internal state, and the cost of processing blocks that are still in transit through the consensus pipeline during reconfiguration. Additionally, as nodes in higher-up levels receive data earlier compared to the lower levels of the tree, it means that the time it takes for the system to begin proposing new blocks could be dependent on the level where the next configuration's leader is located. This logic can be applied to the tree as a whole: the more nodes that are displaced from lower levels to higher ones in a reconfiguration, the higher the likelihood that the system will have to wait for them to complete the procedure.

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

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

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

Lastly, we provide a formalization of the normal case operation of this algorithm in Algorithm 1. Lines 7 – 15 provide the leader's propose functionality, assuming that the `Propose` function receives an already validated block of client transactions. Lines 16 – 28 provide the remaining replicas' proposal handler logic and its steps for ensuring safety. Note that we must preemptively check pending proposals in line 18 for proposals that are out-of-order in a pipeline batch, but we can also proactively check them after reconfiguration on line 35 for proposals that are out-of-order between configurations. The function `ProcessPendingProposals` follows a similar logic to what is seen in lines 25 – 34, however for all the proposals belonging to the *pending_proposals* queue and in-order. We can note that this formalization is simplified by decoupling reconfiguration from the *Pacemaker* [19], which is further expanded on in Section 3.4.4.

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

3.4 Implementation

3.4.1 Challenges. To adjust Kauri to our approach, we must address the following challenges: *i)* Defining an encapsulated and standardized solution for both schedules and topology trees to facilitate intuitive schedule definition and fast extraction of relevant data; *ii)* Adapting the protocol's messages and message handlers to enable the algorithm to concurrently use connections related to different configurations. *iii)* Modifying Kauri's original Pacemaker in order to distinguish planned reconfigurations from forced reconfigurations. Additionally, we need to define the instant where planned

Algorithm 1: Commutation to Next Configuration

```
// Local state for a replica at the start of the protocol
system_trees ← {T_0, T_1, ..., T_n}
current_tree ← T_0
proposer ← GetRoot(current_tree)
child_peers ← GetChildren(current_tree)
lastCheckedBlockHeight ← 0
pending_proposals ← ∅
// Proposal function for leader
function Propose(block):
    if proposer == GetReplicaId() then
        Broadcast(child_peers, PROPOSE(block))
        Vote(block)
        lastCheckedBlockHeight ← block.height
        if block.height == current_tree.target then
            Reconfigure()
        end
    end
// Proposal handler
function ReceivePropose(peer, proposal):
    prop_tree ← system_trees[proposal.tid]
    ProcessPendingProposals()
    // Validate if correct proposer and correct parent
    if (GetRoot(prop_tree) ! = proposal.proposer) ∨
     (GetParent(prop_tree) ! = peer) then
        return
    end
    // Verify block height
    if proposal.block.height ≤ lastCheckedBlockHeight then
        return
    end
    // Check if we have block's causal history
    if (prop_tree ! = current_tree) ∨
     (proposal.block.parent.height ! = lastCheckedBlockHeight)
     then
        pending_proposals ← pending_proposals
         ∪{proposal}
        return
    end
    // Update internal state
    Validate(proposal.block)
    Broadcast(child_peers, proposal)
    Vote(proposal.block)
    lastCheckedBlockHeight ← proposal.block.height
    if lastCheckedBlockHeight == current_tree.target then
        Reconfigure()
        ProcessPendingProposals()
    end
// Transitions to next configuration in the schedule
function Reconfigure():
    current_tree ← NextTree(T_0)
    proposer ← GetRoot(current_tree)
    current_tree.target ← lastCheckedBlockHeight +
     current_tree.duration
```

reconfigurations are triggered such that protocol safety is ensured throughout and after the reconfiguration process. *iv)* Enhancing Kauri's out-of-order message handling so that it takes into account messages that may originate from future configurations.

These challenges are addressed in the following sections.

3.4.2 Standardized Trees and Schedules. Every replica in the system keeps a data structure in which it will store the currently in-use schedule. This structure is instantiated on system startup, as currently, the protocol utilizes a pre-defined schedule. However, implementation-wise, when reconfiguration is triggered, the system is able to edit the schedule safely throughout reconfiguration, thus allowing for the possibility of the protocol interacting with external components to recompute the schedule at runtime. Upon startup, the system can either use a default simple rotation schedule with N trees or extract the schedule from a dedicated file. In the file, each line specifies the fanout, pipeline stretch, and participant order for a tree topology in the schedule. Each tree is assigned a Tree ID (TID), either based on the root of the tree (for the default simple rotation schedule) or the order in which the trees appear in the file (if the schedule is extracted from a file).

We define a *Tree* structure, which represents a tree topology objectively for each replica, tracking properties such as the TID, fanout, pipeline stretch, and the tree array (the order of replicas within the tree). This structure is serializable to allow sharing of trees through the network. Additionally, we define a *TreeNetwork* structure, which acts as a wrapper for the Tree structure and calculates relative replica data for the corresponding topology upon instantiation, such as parent and child peers, the number of children in the replica's sub-tree (for aggregation purposes), and more.

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

- First, protocol messages must now include the TID of the sender's current configuration (for proposals) or the TID of the message it is replying to. Recipients can use this to quickly identify causality and verify whether the sender is the expected source of the message. For example, in the algorithm, proposals are only relayed by parents to their children. Therefore, if a replica receives a proposal, it can validate the message by checking both: i) whether the proposal originates from the proposer of the tree with the matching proposal TID, and ii) whether the sender is the parent of the replica in the tree with the corresponding message TID (Algorithm 1 line 19).
- All message handlers now extract the context of the configuration used to communicate the corresponding message. This is to ensure proper validation and routing of protocol message replies. The process is facilitated by the *TreeNetwork* abstraction, and can be exemplified in Algorithm 1 line 17, where the message's context can be extracted from the *system_trees* map by utilizing the TID contained in the proposal message.

3.4.4 Triggering Reconfigurations. Both our prototype and Kauri utilize what is known as a Pacemaker [19] to handle liveness and

fault detection in the protocol. Implementation-wise, it is the Pacemaker that keeps track of the current configuration and who the current proposer is, and it is also through the Pacemaker that we trigger a reconfiguration in the system.

In normal case operations, planned configurations can be executed by comparing either i) if the proposed block's height reaches the target of the current configuration, in the case of a proposer (Algorithm 1, lines 12 – 13); or ii) if the received proposal block's height reaches the target of the current configuration, in the case of a non-proposer replica (Algorithm 1, lines 33 – 34). This is done after the block is validated and voted for in both cases, maintaining safety, but before the block has been decided, increasing concurrency.

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

3.4.5 Out-of-Order Message Handling. While Kauri's original implementation includes detection and handling mechanisms for pipelined blocks that might arrive out-of-order, it must be adapted to take into account that i) an out-of-order block might be the one to reach the target of the current configuration and ii) that blocks can be out-of-order due to being from a future configuration. By defining a queue for the pending out-of-order blocks at the proposal handling step, we can process them in order. This function can be safely executed at any time in the protocol, although the key moments where it needs to be called are right before we process a new proposal (Algorithm 1 line 18) and right after a reconfiguration (Algorithm 1 line 34). In a worst-case scenario, we may have an entire pipeline stretch of proposals from the next configuration pending, which can all be processed in order in a single execution of our out-of-order message handler.

3.5 Execution Example

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

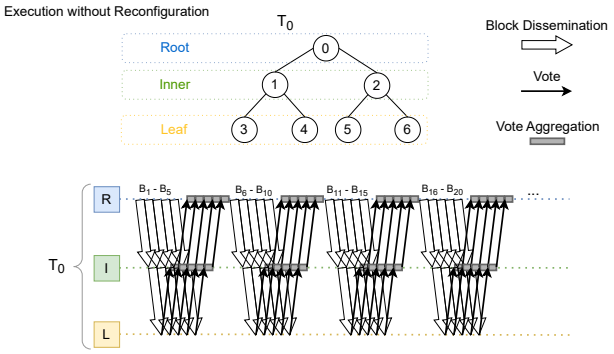

Fig. 2. An example of Kauri's stable leader pipelined execution.

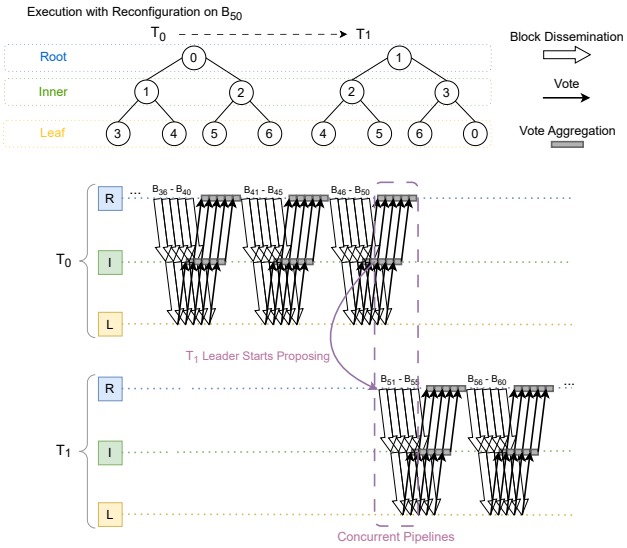

Fig. 3. An example of dynamic reconfiguration.

To exemplify a rotation with our optimised approach, we provide Figure 3. We instantiate this execution with our simple rotation schedule (Figure 1), and additionally, reconfiguration happens every 50 blocks, meaning that T_0 is reconfigured on block B_{50}. In this case, once nodes receive the disseminated block B_{50} from the root, they can start participating in tree T_1's consensus while simultaneously deciding the previous configuration's remaining blocks. This way, the pipeline is finished in the previous reconfiguration while the pipeline in the new configuration is filled, adding concurrency to our reconfiguration process. We highlight the reconfiguration of the node that is T_1's leader with the purple arrow. The gained concurrency is increased the earlier the next configuration's leader receives

the last block of the current configuration. It is important to keep in mind that in larger systems, nodes within the same layer experience greater divergence in message communication times (which is not represented in these diagrams). This accentuates the need for a more thoughtful approach to the placement of the nodes in consecutive tree rotations to better exploit concurrent configurations.

4 Evaluation

In this section, we present various cases of interest that enable us to analyse the impact of our dynamic reconfiguration in Kauri. We establish Kauri's original performance as the baseline and analyse whether or not our reconfiguration mechanism exhibits detrimental costs that would justify picking a stable leader over our rotating leader solution.

4.1 Experimental Setup

This experimental evaluation was conducted through the use of Kollaps [20], which permits us to emulate the different characteristics of a distributed network, such as the latency and the bandwidth of the links between the nodes. Every experiment was conducted with two physical machines connected in a way that communication costs between them are insignificant when compared to the emulated topology. The workload we provided puts computational resources close to saturation, meaning that processing time is the bottleneck of the system. Every network utilized defines its network properties at the link level. Each participant has a link to every other participant. Implementation-wise, network communication leverages concurrency to improve message dissemination, meaning that multiple links can be used throughout the process to reduce the sending time of blocks for consensus. It is possible to saturate node communication links by defining a topology where all of a participant's traffic is routed locally through a switch, making it so that a node's link to its switch is a point of contention and the bottleneck for communication. Since resources are already near saturation at the computational level, we choose networks where each participant has a dedicated link to every other participant, avoiding the interference and contention that would occur on both resources when using a bottlenecked switch-based topology.

To simplify our experimental evaluation, we assume that all the trees in the schedule have an equal target duration of k blocks. Since all reconfigurations will occur every k blocks, systems with higher throughputs will be reconfiguring more frequently.

4.2 Homogeneous Networks

The goal of running experiments in a homogeneous network environment is to limit the number of factors that can influence the throughput of the system throughout the process of reconfiguration. We study the impact of the following three factors in the performance of our dynamic reconfiguration: i) the height of the trees the system uses; ii) the displacement of the nodes between consecutive configurations and iii) the frequency at which the system reconfigures. For all the experiments run, unless stated otherwise, we attributed a latency of *50 ms* and a bandwidth of *750 Kbp/s* to all the connections used between the participants. The selected bandwidth makes it so that the sending time is still a significant factor for the execution of our algorithm, even if dissemination is parallelized and optimized.

4.2.1 Tree Height. For this experiment, we evaluate how the height of the trees being used can affect our reconfiguration procedure. In all the executions presented, our trees have a target duration of $k = 300$ blocks and we utilize a simple rotation schedule (Figure 1). To accurately represent the expected load of a Kauri execution, the pipeline stretch of each scenario was adjusted taking into account the height of the trees used for consensus. We analyse the following scenarios: *i)* Height of $h = 2$: using $N = 31$ nodes structured into a star topology of fanout $m = 30$. This scenario corresponds to HotStuff's [3] topology; *ii)* Height of $h = 3$: using $N = 31$ nodes structured into trees of fanout $m = 5$; *iii)* Height of $h = 5$: using $N = 31$ nodes structured into trees of fanout $m = 2$.

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

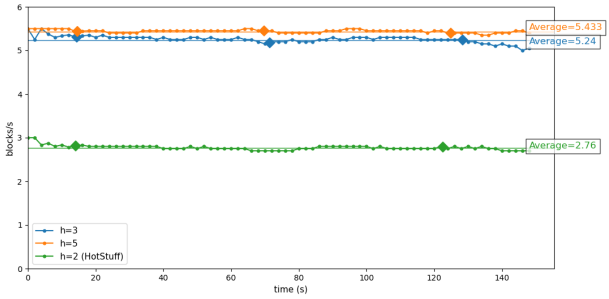

Fig. 4. Throughput (in blocks per second) over time of the three different scenarios where the tree heights are varied.

In these scenarios, the processing power available for each node is the bottleneck of the execution, further accentuated when the fanout is increased. That said, a star topology execution is the scenario with the worst performance, even when considering the fact that the implementation can make use of the concurrency offered by our topology to reduce HotStuff's heavy sending time. The results of this experiment are represented in Figure 4. In the graph, the different scenario executions are aligned so that their first reconfiguration occurs at the same point in time. Additionally, each different scenario has its overall average throughput represented by a horizontal line.

With this experiment, we can deduce that the overall throughput of the different scenarios follows what was expected regarding the fanout utilized in each of the executions: the scenario that used a HotStuff-like star-based topology for its trees ($h = 2$) has the lowest throughput, as the resources available for a single node are the bottleneck in this experimental setup, being close to saturation. Meanwhile, both executions with higher-depth trees present similar throughputs which are better than HotStuff, as the tree topology balances consensus' workload. Additionally, as the pipelining techniques used were adjusted according to the tree's height, these compensate for the added latency of the trees. The slight difference in performance between trees with $h = 3$ layers and $h = 5$ layers

can be justified due to the differences originating from the workload distribution: the trees with fanout $m = 2$ were able to more evenly distribute consensus workload. Although this is a valid strategy for distributing workload in this experiment, it is not applicable to every context as real-world scenarios have to take into account the drawback of more nodes being assigned to internal nodes in the tree, which increases the difficulty of finding a robust tree.

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

4.2.2 Node Displacement. For this experiment, we evaluate the impact that the displacement of nodes between two consecutive configurations has on the dynamic reconfiguration's performance. For our executions, we use trees with $N = 31$ nodes and a fanout of $m = 5$ (consequently, $h = 3$). Each tree now utilizes a target duration of $k = 100$ blocks to reconfigure more frequently. To measure the impact of node displacement in our reconfiguration process, we compare the performance of the reconfiguration mechanism between two different schedule strategies, namely: *i)* **Root-Child Switch Schedule**. This schedule oscillates between two trees, where the root of the next tree is a child of the root of the current tree. It is expected that this schedule has a reduced impact on the reconfiguration process because by only switching the root with a direct child, the latency of the switch is reduced (as nodes only jump one layer in the tree) and the disruption on the pipeline will be kept to a minimum; *ii)* **Internal-Leaf Switch Schedule**. This schedule oscillates between two trees, where the internal nodes of a tree become the leaf nodes of the following tree, and vice-versa. In this schedule, the root of the tree will always be swapped with a leaf node. It is expected that this schedule has a bigger impact on the reconfiguration process, as a high amount of nodes jump various layers in the tree. Additionally, we include an execution without reconfiguration (i.e., a stable leader execution) to act as the baseline of the expected stable performance. The results are displayed on Figure 5.

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

Secondly, as expected, the Internal-Leaf Switch schedule applies more strain to the throughput of the system due to the higher displacement of nodes. This can be seen by the fluctuations of the throughput throughout the execution when compared to Root-Child Switch schedule's more stable throughput. Even then, for such a high displacement of the pipeline structure, the reconfiguration costs are relatively low. This aligns with what was witnessed in the rotating schedule used in Figure 4, where the reconfiguration costs were kept at a minimum even when a moderate amount of nodes

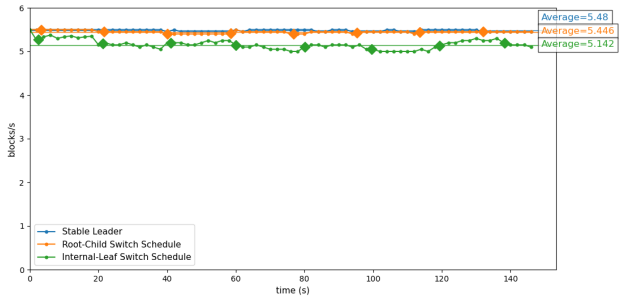

Fig. 5. Throughput (in blocks/s) over time between two systems with different schedules, compared to a system that has no reconfiguration.

had to switch their respective parent and children nodes during reconfiguration.

Thus, for schedules that follow Internal-Leaf Switch's behaviour, where most of the nodes change roles upon reconfiguration and the pipeline suffers a bigger disruption, reconfiguration is still feasible without heavy costs. However, we would like to note that the throughput pattern witnessed here is noisier than expected and did not manage to reach the other executions' stable throughput. It is easy to suspect that, if the system reconfigures faster than the throughput stabilizes after the reconfiguration's disruption, then inherently the system will never be able to reach stable performance. This can be further accentuated by the fact that the system's resources are near saturation. For this reason, we provide a follow-up test to this schedule type in Figure 6, where the computational load is alleviated and where we give more time between reconfigurations to confirm reconfiguration recovery time.

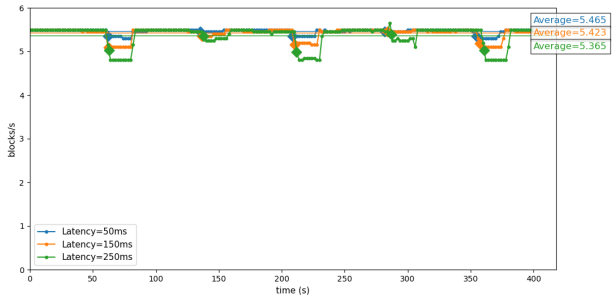

Fig. 6. Throughput (in blocks/s) over time in Internal-Leaf Switch schedule executions where we vary the latency.

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

blocks. As seen in Figure 6, recovery time for reconfiguration is consistent across all networks, with higher latency networks having a stronger throughput impact. Even then, the overall average throughput of all executions is largely unaffected, maintaining similar values. We also note that certain reconfigurations present in this schedule end up having less impact than others, which highlights how important it is for schedules to not only have in mind network properties but also node workload and computational power.

4.2.3 Reconfiguration Frequency. Finally, we study the impact that the frequency of reconfiguration can bring to the performance of the system. We once again use a simple rotation schedule and a system with trees of $N = 31$ nodes with a fanout of $m = 5$. This time, however, we evaluate the performance of our system for four distinct k values, namely: *i)* $k = 1$: A system that rotates leader every block, emulating LSO algorithms like HotStuff [3]; *ii)* $k = 25$: A system that rotates every 25 blocks; *iii)* $k = 50$: A system that rotates every 50 blocks; *iv)* $k = \infty$: A system where the configuration is not changed (Stable Leader). The results are shown on Figure 7. We include the execution that uses HotStuff's topology from Figure 4 for comparison purposes.

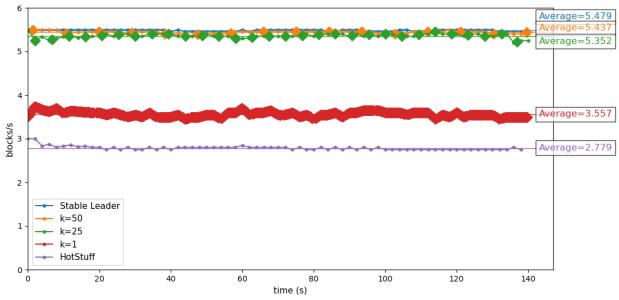

Fig. 7. Throughput (in blocks/s) over time between four different systems with different reconfiguration frequencies and HotStuff.

Once again, for the homogeneous network selected, reconfiguration following the rotation schedule does not seem to provide any noticeable costs for values where k does not interrupt the pipeline. In these cases, there are negligible throughput losses when the reconfiguration rate is higher. Meanwhile, when $k = 1$, Kauri suffers a loss of throughput as, by reconfiguring every block, the system can not make use of its pipelining techniques to compensate for the latency derived from its tree topology. Even then, Kauri still gains throughput in this context over the HotStuff topology seen in Figure 4, as it more evenly distributes the load across various participants. This means that a LSO approach to Kauri can in fact outperform HotStuff when combined with our dynamic reconfiguration.

4.3 Heterogeneous Networks

We can infer the utility of our solution in a WAN by once again utilizing Kollaps, however this time with a careful definition of various network links that could represent a real-world deployment of a permissioned blockchain application. A variety of works take into account that many permissioned blockchains are usually deployed as a data center network, meaning that most participants can be grouped and approximated to different clusters. [18, 21, 22]. Given this, for our heterogeneous experimental evaluation, we define a simplified version of a cluster-based environment for the definition of our network links.

We maintain a bandwidth of 750 *Kbp/s* for all links in this setup. This is due to how computational resources are already near saturation, meaning that for our emulated environment, maintaining the previously used bandwidth provides clearer results for interpretation. The most significant aspect of this heterogeneous network definition is not the absolute values chosen for the network properties (which are influenced by and adjusted to our experimental infrastructure), but the relative values between the cluster communication links. The asymmetry brought forth by the latency discrepancies between the clusters is enough to showcase the impact that a heterogeneous network may have on system execution. With this, we can say that network properties for the connections between the three different clusters A, B and C are set so that: *i)* Intra-cluster latency is reduced for all clusters, meaning that trees should use same cluster links as much as possible; *ii)* Latency between A and B is moderate, same with latency between B and C; *iii)* Latency between A and C is relatively bad, meaning that trees should avoid using links between these two clusters.

We once again use a system with $N = 31$ nodes structured into balanced trees of $m = 5$. Nodes are distributed so that the clusters are balanced. To evaluate the impact of a smart tree schedule in this heterogeneous network setting, we establish the following three schedule executions: *i)* **Rotation Schedule**: the same rotation schedule defined in Figure 1; *ii)* **Randomized Schedule**: a schedule of N trees where trees are completely randomized, akin to what Kauri originally used for its bucket-based construction; *iii)* **Informed Schedule**: a schedule of N trees that avoid weak links and prioritize intra-cluster communication.

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

From Figure 8, we can infer that a random schedule has the worst performance in the heterogeneous network provided. This is because with already 3 different kinds of clusters, the network already has enough variety to increase the probability of inner nodes having high latency links. These network asymmetries can heavily increase the wait experienced to aggregate votes and therefore in consensus. Simultaneously, a rotation schedule provides a performance that can be compared to a random schedule, however bringing forth more consistency. This is because throughout execution tree performance will often only slightly diverge, as this type of schedule maintains a moderate portion of past links between configurations. Lastly, we note that an informed schedule ends up having better performance than the other two schedules throughout the entire execution. In

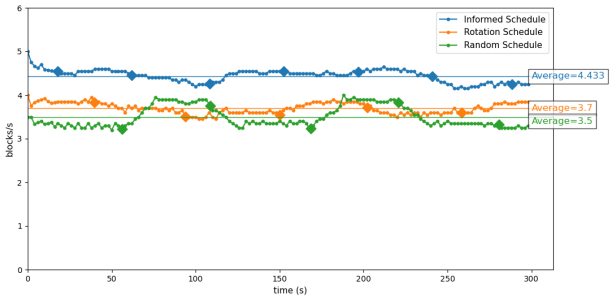

Fig. 8. Throughput (in blocks/s) over time of three different schedules in a heterogeneous network.

this case, trees can be set up to avoid weaker links while still shifting around the leadership role between configurations. In theory, informed schedules can be further adjusted in different heterogeneous contexts, such as increasing the pipeline stretch according to the current tree's expected remaining time from its links' latency.

5 Conclusions and Future Work

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

Overall, we were able to confirm that Kauri is compatible with a rotating leader policy and that with our solution we can hope to obtain even better scalability in real-world scenarios, whilst obtaining the full benefits of a rotating leader consensus protocol. For future work, we aim to enable our approach to recompute schedules at runtime. By defining tree performance metrics and leveraging external systems to measure node correctness and/or network performance, our approach would allow tree-based consensus with a rotating leader policy to perform well in WANs, while also adapting to environmental changes during protocol execution.

Acknowledgments

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

References

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

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

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

[4] R. Neiheiser, M. Matos, and L. Rodrigues, "Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.* Association for Computing Machinery, 2021, pp. 35–48.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] P. Gouveia, J. a. Neves, C. Segarra, L. Liechti, S. Issa, V. Schiavoni, and M. Matos, "Kollaps: decentralized and dynamic topology emulation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020.

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

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