Dynamic Adaptation of Tree-Based Blockchain Protocols

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

Fábio Gomes — 103614*
gabriel.gomes@tecnico.ulisboa.pt

Advisors: Professors Luís Rodrigues and Miguel Matos

Abstract Some blockchain protocols use dissemination and aggregation trees to increase the scalability and performance of the system. Previous systems based on this approach have been mainly focused on ensuring liveness, i.e., the trees used by the protocol are chosen to ensure that the protocol can terminate. This requires that at least one of the trees used by the protocol allows enough correct processes to exchange information reliably (such tree is denoted *robust*). In this work we aim at extending these systems with mechanisms that: i) promote the selection of robust trees and ii) among multiple alternative robust trees, promote the selection of trees that maximize the performance of the protocol (by selecting reliable participants with more networking and processing bandwidth as inner nodes and by organizing the tree to minimize the latency of the dissemination and aggregation tasks). We propose an architecture that supports the dynamic selection of trees based on the node resources, network conditions, and past behaviour of participants (i.e., reputation) to improve the performance of the blockchain system. We also propose mechanisms to ensure that correct nodes can agree on which tree to use and that prevent malicious nodes to bias the system towards the use of less performant or non-robust trees. We plan to assess our architecture by building and evaluating an extended version of the Kauri blockchain protocol.

Keywords — Distributed Systems, Byzantine Fault Tolerant Consensus, Adaptive Scheduling

*I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa (https://nape.tecnico.ulisboa.pt/en/apoio-ao-estudante/documentos-importantes/regulamentos-da-universidade-de-lisboa/).

Contents

1 Introduction

The blockchain is an abstraction that aims at support the coordination among multiple participants, that do not necessarily trust each other, by keeping a totally ordered, immutable, sequence of records that can be updated in a distributed manner. Blockchains have been used to support several distributed applications, such as distributed finance [1, 2], decentralised markets [3, 4], provenance tracing of goods [5, 6], among others. At the core of a blockchain system there is a distributed consensus protocol [7], that allows participants to agree on the total order of records. This protocol needs to be fault-tolerant and should be able to tolerate arbitrary behaviour from faulty (potentially malicious) participants, what is known as Byzantine Fault Tolerance (BFT) [8].

In this work we address permissioned blockchains [9], where the set of nodes that participate in consensus is known. There are several approaches to implement Byzantine fault-tolerant consensus in this setting, including *leaderless* [10–12] and *leader-based* [13] approaches. In our work we focus on leader based protocols. Several leader based protocols have been proposed in the literature, mostly inspired by the PBFT protocol [13]. PBFT requires nodes to engage in all-to-all communication rounds, a communication pattern that exhibits n^2 message complexity, which limits the scalability of the system. Subsequent works have proposed protocols that are more scalable, HotStuff [14] uses only one-to-all and all-to-one communication rounds, reducing the message complexity from quadratic to linear, Kauri [15] uses dissemination and aggregation trees to distribute the load and ensure that each node only interacts with a number of participants that is logarithmic with the system size.

In any of these protocols, the performance of the system depends on the characteristics of the leader. If the leader is faulty, a new leader needs to be elected to ensure progress. Even a correct leader, when resource-constrained, may be a bottleneck that limits the performance of the system. This is aggravated in tree-based protocols, since the performance depends not only on the leader but also on the properties of the inner nodes of the tree. If the leader or the inner nodes exhibit poor performance, the system may benefit from reconfigurations. Finally, even when a configuration is performing well, reconfiguration can be useful to distribute the load and improve fairness [16].

The state of the art presents many solutions for the adaptation of blockchains [17–19]. In spite of this, the works present in the literature seldom consider the specifics of tree-based blockchain, where there exist non-leader nodes which can affect the robustness of a configuration. The SmartTree [20] protocol, which discusses the adaptation of trees, lacks a detailed presentation and evaluation of results.

As such, in this work we are interested in studying techniques that can drive the reconfiguration of tree-based Byzantine consensus protocols, such that reliable and performant configurations are more likely to be chosen than configurations that include faulty or under-performant processes as inner nodes. Such mechanisms can offer better *chain quality* [21], lower latency, and higher throughput. We aim to use monitoring information and reputation mechanisms to assign scores to nodes, links, configurations, and, ultimately, to sequence of configurations. These scores will help to find suitable sequence of configurations that match resilience and performance goals for the system operation.

The remainder of this report is structured as follows: Section 2 briefly explains our goals. In Section 3 we explore the background and related work. Section 4 discusses the approach we take when designing our architecture, while Section 5 proposes an initial implementation. Section 6 expands on how we intend to evaluate our proposal. Section 7 presents the schedule for our work and Section 8 concludes the report.

2 Goals and Expected Results

The project has the following goals:

> Goals: Our work seeks to create an architecture enabling the dynamic adaptation of tree-based blockchain protocols. This architecture should explore different performance indicators along with the properties of node interactions in the tree topology to find and evaluate suitable configurations, in order to schedule fair and performant sequences of configurations.

The project is expected to produce the following results:

> Expected Results: The work will produce 1) an architecture proposal, 2) an implementation of the architecture specification, and 3) an evaluation of the architecture implementation.

3 Background and Related Work

In this section we introduce the background required to develop our work and survey existing work. At the end of the section we offer a critical analysis of the existing work, and motivate the need for techniques that support the dynamic selection of dissemination and aggregation trees in the context of blockchain consensus.

3.1 Blockchain

A blockchain is a decentralised, persistent, and totally ordered sequence of records that is maintained, in a fault-tolerant manner, by multiple participants. As the name suggests, records are aggregated in *blocks* that are linked to each other (chained). Cryptographic techniques are used to ensure that the task of tampering with blocks (or links) added to the blockchain is infeasible, making the blockchain tamper-proof.

Since fault-tolerance is a key aspect in the operation of a blockchains, the task of adding new blocks to the blockchain is typically not assigned to a single participant. Instead, multiple participants are allowed to add new blocks to the blockchain. If two participants attempt to extend the chain concurrently, two different blocks may be created with the same predecessor in the chain, creating a *fork*. To avoid forks, participants must run a distributed consensus protocol to agree on a single sequence of blocks.

Blockchains were invented alongside the Bitcoin cryptocurrency system [7], where the blockchain is used to record all transactions that occur in the system. In the Bitcoin network any node can join the system and attempt to add blocks to the chain. Such system is called a *permissionless* blockchain. In other systems, only a few selected participants are allowed to propose blocks; these system are called *permissioned* blockchains.

3.1.1 Permissionless Blockchains

Permissionless blockchains have been proposed by Nakamoto [7] and are commonly associated with their use in systems such as Bitcoin and Ethereum [22]. In these blockchains, any node can propose a new block to be added to the chain. The proposer disseminates the block in the network. A participant that has not proposed a block with the same predecessor and sees a block from a different proposer, refrains from proposing a competing block, and starts working on a new block to extend that one. If no two blocks with the same predecessor are proposed concurrently, this mechanism prevents forks from occurring. If two participants concurrently propose different blocks with the same predecessor, a fork occurs. Eventually, one of the forks will evolve faster than the other(s). Blocks in the longest branch are preserved and blocks in other branches are discarded.

These protocols do not provide *finality*. Informally, finality states that, when a block is added to the blockchain, it can no longer be discarded. In fact, when using the protocol above, there is no point in time where one can guarantee that a block will never be discarded, because a longer branch may always be found in the future (although the probability of finding a longer branch decreases as the length of a branch grows).

Permissionless blockchains include mechanisms to ensure that the number of participants that a single user controls is proportional to the resources that user commits to the system. One of such mechanisms is *proof-of-work*, where a participant must solve a crypto-puzzle before proposing a block, limiting the rate at which a participant can propose blocks. Because the time it takes to solve the crypto-puzzle is not deterministic, this

mechanism also reduces the likelihood of having two participants proposing a block with the same predecessor concurrently.

3.1.2 Permissioned Blockchains

In a permissioned blockchain, only selected participants are allowed to propose blocks. Therefore, at a given time, the number of participants in the consensus protocol is fixed and known. This allows the use of classical *Byzantine Fault Tolerant* (BFT) consensus protocols that have been proposed in the literature and that can provide finality. One limitation of BFT protocols is that they scale poorly with the number of participants. Therefore, permissioned blockchains are only suitable for system where the number of participants is small.

3.1.3 Fairness

In many blockchain applications, the order by which transactions appear on the chain can be relevant. For instance, a user may gain benefits if her transaction appears before a similar transaction from other users. It is therefore undesirable to let most blocks be proposed by the same participants, given that this gives that participant an unfair advantage over deciding how transactions are ordered. Most blockchain protocols have mechanisms that attempt to give any correct participant a fair opportunity to produce blocks. For instance, in leader-based BFT protocols, this can be achieved by rotating the leader role among the consensus participants.

3.1.4 Chain Quality

If the participant that proposes a block is faulty, it may include bogus transactions in a block. A relevant property in the operation of a blockchain is the *Chain Quality* [21], which captures the fraction of adversarial transactions contained in a continuous portion of the blockchain. One way of improving the chain quality is to identify participants that propose blocks with bogus transactions and to design mechanisms that decrease their chance of producing blocks.

3.2 Byzantine Fault-Tolerant Consensus

Byzantine Fault-Tolerant (BFT) consensus is a distributed system problem that consists in ensuring that a set of correct processes agrees on a common value even in the presence of a minority of faulty nodes that can exhibit arbitrary behaviour, including malicious behaviour. BFT Consensus is defined by the following properties [23]:

1. *Termination*: Every correct process eventually decides some value;
2. *Validity*: Every value decided by a (correct) process has been proposed by some process;
   (a) *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;
   (b) *Strong Validity*: If all correct processes propose the same value, v, then no correct process decides a value different from v; otherwise, a correct process may only decide a value proposed by some correct process or the special value $\square$;
3. *Integrity*: No correct process decides twice;
4. *Agreement*: No two correct processes decide differently.

The validity property is of note, as two formulations arise in BFT consensus, dividing algorithms into weak and strong. The main difference being in how they handle Byzantine faults, in Weak Consensus, decided values are only guaranteed to come from a correct process when every process is correct and proposes the same value. While, in Strong Consensus, decided values are guaranteed to originate from correct processes, else a special value $\square$ is decided.

BFT consensus protocols may adopt different network models. Since a deterministic algorithm to solve consensus in asynchronous networks is impossible [24], many systems adopt a partially synchronous model

[25] in order to ensure termination, while agreement can be provided even in asynchronous conditions. In a partially synchronous network there may be an unstable period where messages exchanged can be indefinitely delayed. However, after an unknown Global Stabilisation Time (GST), the latency on correct processes' messages is bounded by a known Δ.

In the following, we discuss some of the most relevant algorithms that have been proposed to solve BFT consensus.

3.2.1 Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) [13] is a BFT consensus algorithm designed for practicality and efficiency in asynchronous systems, where messages can be delayed an indefinite amount of time. PBFT assumes an asynchronous network of N nodes where $f < N/3$ faulty nodes can be tolerated.

The protocol is structured such that, at a given time, one replica is the *primary* (or leader), while the remaining serve as *backups*. The mapping of participants to roles (i.e., which node serves as leader) is called a configuration (or *view*). The configuration may change over time. Typically, in PBFT a view change occurs when the primary is suspected to have failed. Primaries are pre-assigned to each view in a round-robin manner, such that $P = V \mod N$, where P is the ID of the primary and V the number of the view.

The algorithm begins when the primary receives a request from a client, which is forwarded to the backups in a *pre-prepare* message, containing the request message itself, along with a sequence number, view number, and a digest of the request message, all signed by the primary. If the pre-prepare message is accepted, replicas sign and multicast a *prepare* message to all other replicas, containing view number, sequence number, replica ID and request message digest. After a replica receives $2f$ valid prepare messages from different backup replicas, it can execute the request and *commit* the result to the client in a signed message. The client accepts a result after $f + 1$ equal results have been received, meaning that at least one correct process commits the result.

In order to detect the failure of the primary, each replica has a timeout after receiving a valid client request, after which it multicasts a *View-Change* message. When the node that serves as primary of the following view receives $2f$ view change messages, it multicasts a *New-View* message, making the new configuration active.

3.2.2 HotStuff

HotStuff [14] is a BFT replication protocol that avoids the quadratic message complexity of PBFT (that results from the all-to-all exchange of protocol messages such as the *prepare* message). It is divided into three vote collection phases: *prepare*, *pre-commit* and *commit*, in addition to a *decide* phase. In the *prepare* phase, the leader chooses a command to append to the log. The other replicas are then tasked to vote on whether to accept this command or not. In the *pre-commit* phase, the replicas confirm a quorum has accepted the proposal. In the *commit* phase, each replica can safely lock the value of this proposal so as to not accept any future conflicting proposal. Finally, in the *decide* phase, the proposal is committed and stored on the log.

Phases are structured so communication is held directly between the leader and the remaining replicas, in what is called a star communication pattern. Leaders relay messages to the remaining replicas and collect their votes along with partial signatures, which are combined into *Quorum Certificates* and used in the succeeding phases of the protocol as proof of correct behaviour. Protocols like PBFT, which use all-to-all communication (or *cliques*), exchange a number of messages that is quadratic with the number of nodes in the system. HotStuff, by leveraging the star pattern, greatly improves the complexity of its communication scheme, as the number of exchanged messages scales linearly with the number of participants.

Additionally, each phase has a waiting period which can be interrupted by a *NextView* interrupt, signalling a replica to start the next view. One of the main concerns of HotStuff is achieving a *Linear View Change*, where the number of messages required for nodes to converge on a view-change is also linear with the number of processes. By lowering the cost of changing leaders, HotStuff improves on the *chain quality* and *fairness* provided by previous protocols.

Chained HotStuff is a streamlined variant of the base protocol, where there is only one kind of message and each command is proposed in a different view. Upon conclusion of the *prepare* phase, the proposal's quorum certificate is passed to the next view's leader, which adds to this a new proposal and broadcasts the message. The remaining replicas then vote for the old proposal's *pre-commit* phase and the new proposal's *prepare* phase

simultaneously. This process is continued into the next phases, leading to one message per view, each containing information on four proposals, each in a different phase.

3.2.3 Kauri

Kauri [15] extends HotStuff with the tree communication pattern, replacing HotStuff's star pattern. In a star communication pattern, the leader is responsible for disseminating and aggregating messages to and from the remaining nodes, as well as validating the cast votes. This centralisation poses a problem for scalability, as the processing power of the leader limits the size of the network running the protocol. In contrast, the tree pattern proposed by Kauri distributes the load from these operations to all the inner nodes of the tree. This comes at the cost of latency and, consequently, throughput, as messages need to travel the total height of the tree, instead of being directly sent to/from the leader. Additionally, the tree pattern introduces new points of failure, as inner nodes can interfere with the process of disseminating and gathering information.

The use of dissemination and aggregation trees causes the leader to remain idle while information is propagated in the tree. This adds latency to the protocol and causes poor resource utilisation. To mitigate this effect, Kauri uses the leader's idle time between block proposals to optimistically initiate new consensus instances, a technique known as *pipelining*. Furthermore, Kauri employs an optimisation akin to chained HotStuff; after collecting the votes related to a given phase of consensus, the protocol merges the information of various phases across consensus instances into a single message.

In HotStuff, when the network is stable, termination is guaranteed in a configuration where the leader is correct. Thus, at most $f+1$ view changes are required to find a configuration that ensures termination. In Kauri, it is not enough to have a correct leader to ensure termination: the selected tree must ensure that a Byzantine quorum of correct participants can reliable exchange information. A tree that satisfies this property in called *robust*. The presence of Byzantine participants as inner nodes of the tree may prevent the tree from achieving robustness. Because the number of possible trees is factorial, and only a small fraction of these trees are robust, if no mechanism is in place to select the trees used by the algorithm, a large number of view changes may be needed to ensure termination. Kauri implements a strategy to select configurations that ensures termination after a linear number of view changes. This is achieved by sorting all processes into bins, with each bin able to holds as many processes as the amount of internal nodes, and constructing a tree per bin with that bin's processes as internal nodes. This results in m bins, m being the fanout of the tree. If the amount of faulty nodes, f, is less than the tree's fanout, a solution will be found within m reconfigurations. Else, Kauri falls back to HotStuff's network pattern, incurring Hotstuff's performance drawbacks; however, guaranteeing a solution is found within $m + f + 1$ reconfigurations.

3.2.4 Kauri Extensions

We now describe a number of extensions to the Kauri protocol.

Heterogeneity Awareness To build the dissemination and aggregation trees, Kauri creates a tree with a fixed fanout that is chosen considering an estimate of the network bandwidth and latency that is assumed to be uniform. Given the selected fanout, Kauri divides nodes in different bins and builds each tree by picking nodes from each bin and randomly placing them as inner nodes of the tree. The work in [26] studies the behaviour of different trees in settings where the bandwidth and latency among nodes is heterogeneous. The study shows that a topology-aware approach to build trees has the potential to improve Kauri's performance significantly. The same work also suggests some strategies that can be used to build trees that yield good performance.

Fast Reconfiguration Kauri uses a stable leader approach, where a leader change only occurs when the current leader is suspected to be failed. In such setting, the performance of the reconfiguration procedure is not of prime relevance, given that the downtime during reconfiguration is dominated by the time it takes to suspect the leader. In [27], a number of extensions to the reconfiguration protocol used in Kauri are proposed and evaluated, with the goal of reducing the performance penalty of scheduled leader changes such that it becomes viable to implement alternative leader management strategies, such as, for instance, a rotating leader approach. The proposed solution entails defining a k amount of blocks to be delivered by a tree before reconfiguring.

As such, after voting on the k-th block of a given tree, a node can begin transitioning to the next scheduled configuration. Blocks that were proposed in tree T_i finish processing in parallel with the initialisation of the T_{i+1} pipeline; blocks are processed in their respective tree. This way the pipelines from each tree can be fully realised, while the tree's latency and reconfiguration overhead are mitigated by the continuous processing of blocks.

Scheduling Reconfigurations To increase fairness and load-balancing, it is interesting to expand Kauri to support periodic reconfigurations, where the nodes that serve as leader and inner nodes of a tree may be replaced frequently. The work of [28] has started to sketch a number of strategies to achieve this goal. First, it proposes that the reconfiguration schedule is divided in *epochs*, where each epoch consists of a sequence of trees. The system operates by rotating among the trees in an epoch by changing tree after a pre-defined amount of time or a pre-defined number of blocks. To ensure that trees in an epoch have good performance, the work also suggests the use of a reputation scheme, that would help to promote nodes with good behaviour for inner nodes in the trees. To the best of our knowledge, the ideas proposed in [28] have never reached maturity.

3.3 Adaptive BFT Systems

The BFT consensus algorithms described in the previous section are *leader* based, i.e., they require that some process acts as a coordinator of the algorithm to ensure termination.[1] Some protocols assume that, once elected, a leader remains active until it is suspected to be failed. Such protocols are said to use a *stable leader* approach. Other protocols require the leader to be changed periodically, sometimes at every block that is produced, an approach called *rotating leader*. As discussed below, both approaches have advantages and disadvantages. In either case, the strategy to change the leader may be static and defined *a priori*, for instance, using a simple round-robin scheme, or be *adaptive*, where the leader change is driven by the past behaviour of each node and by the observed operational conditions (such as latency, bandwidth, CPU utilisation, etc). We now discuss some relevant adaptive systems.

3.3.1 Carousel

In regular round-robin leader selection, crashed processes pose a major obstacle to performance, as these processes are repeatedly chosen in each iteration of the round-robin, leading to unproductive rounds which greatly impact system latency and throughput. *Carousel* [17] is introduced as a leader-rotation solution which leverages on-chain information to improve system performance by selecting performant, active leaders in each round.

To achieve this, the authors define the property of *leader-utilization*, which states that in crash-only executions, after GST, the number of rounds r for which no honest party commits a block formed in r is bounded. This property ensures that the number of faulty parties elected as leader is bounded; as a result, the number of rounds where honest processes are unable to make progress is limited by a finite number. By satisfying *leader-utilization*, alongside *termination*, *agreement* and *chain quality*, Carousel provides a leader-aware framework for BFT leader selection in crash-only executions (i.e. executions containing no Byzantine failures).

In order to achieve leader-utilization, Carousel employs a reputation scheme which is based on block endorsement. Processes which validate a block B in round r are endorsers of B, processes which do not endorse B do not hold the reputation to be chosen as leaders of round $r+1$. By pooling from the previous round's endorsers, Carousel ensures the leader candidates had not crashed up until that round; this method also biases the leader selection toward more participatory, performant processes, which reflects on the performance of the whole chain. Additionally, in order to provide chain quality, the last f leaders are excluded when selecting a leader from the set of endorsers. This way, a deterministic rule may be applied to leader selection while preventing processes from manipulating the rule to obtain disproportionate representation in the leader role.

When a process does not commit any block in round r, the leader of $r+1$ cannot be elected via reputation, in this case the system falls back on round-robin selection. There are no guarantees on which method a given process will use to select its leader, leader divergence may occur. This could compromise termination, since crashes, delayed messages or adversarial behaviour can lead to leader disagreement. However, termination, leader-utilization and chain quality are satisfied, as the number of rounds r for which no honest party commits a block formed in r is bounded by $O(f^2)$; and at least one honest block is committed $5f+2$ rounds.

[1] There are also *leaderless* protocols [10–12], but these are outside the scope of our work.

When applied to previous round-robin-based systems, such as HotStuff, the authors have shown that Carousel offers better performance in executions containing crashes, since crashed processes did not enter the leader role; as well as in crash-free executions, as better performing leaders are selected.

3.3.2 Hammerhead

Hammerhead [29] proposes a leader scheduling scheme for DAG-based protocols [30] inspired by Carousel to optimise leader-utilization. A property of DAG-based protocols is that they inherently provide chain quality; thus, Hammerhead's specification is only concerned with providing agreement and termination.

The scheduling is divided into epochs initially set up as a fair round-robin schedule, S_0. During the schedule execution, validators which vote on a leader's proposal have their reputation scores increased after a delay that ensures consistency across nodes. After a set amount of leaders, a new schedule, S', is created from a round-robin of the previous schedule, S, and using two sets of nodes, B and G. These sets are of equal size and hold up to f nodes with the lowest and highest reputation scores, respectively. By replacing each B validator with a G validator in the schedule, S' is calculated and takes immediate effect. The schedule switch is a point where agreement and termination are at risk of being violated. Careful application of the schedules is required to uphold agreement. Furthermore, unsynchronised validators may disagree on the leader and be unable to make progress, compromising termination; however, at the Global Stabilisation Time (GST) validators are able to be synchronised, ensuring progress from thereon and preserving termination.

Analogously to the results found in Carousel, when applying Hammerhead to *Bullshark* [31], a reference DAG-based protocol, it was possible to observe performance improvements in both crash-free and crash-only executions without displaying visible throughput degradation with the introduction of faulty processes.

3.3.3 Adaptive Baseline Score-based Election

Adaptive Baseline Score-based Election (ABSE) [32] is an approach to perform adaptive leader elections into BFT protocols. ABSE is aimed at protocols which change leader frequently (such as through round-robin or random selection), providing a generic set of components for a localised reputation system (ABSE data is handled independently in each node and excluded from communication) to enable adaptive leader selection.

In ABSE, every process starts with a base 0 score. The approach relies on rewarding the nodes which promote a decision in any given round r of consensus by increasing their score. In order for a score to have influence on protocol decisions, a certain amount of confirmations, denoted as ct, have to first occur. As such, the election of a leader for round r, is based on the scores of round $r - ct + 1$. When the underlying protocol makes a leader decision, it compares the available ABSE scores with a baseline score and only accepts processes scored above the baseline, if the protocol's chosen leader is not accepted by ABSE, a view-jump is triggered. ABSE defines baseline scores as a function of the current round.

ABSE aims to be a generic approach, that may be applied to many differing existing protocols. Thus many of the components of the approach can be tailored to concrete settings, including the rate at which processes are rewarded for promoting consensus, how a baseline score is derived from the round number or the number of confirmations needed for the scores. In this way, ABSE implementations could cater to protocols which may make leader decisions at different round intervals, or that may require longer confirmation times as not to let adversarial processes undermine honest participants.

The authors have implemented ABSE in HotStuff and DAG-Rider [30]. The observed results show little degradation in base performance, as well as the importance of correct configuration of baselines for correct processes to be promoted as leaders. In regards to performance under faults, the ABSE enhanced protocols provided much greater performance than their regular implementations.

3.3.4 Reputation-based Byzantine Fault Tolerance

Reputation-based Byzantine Fault Tolerance (RBFT) [19] proposes a variant of PBFT which provides a reputation system to the algorithm enabling 1) adaptive primary selection biased towards the most reputable processes, and 2) differentiated discourse rights between processes, where some processes' votes weigh more than their peers'. The system is designed in a way where nodes require expected block issuing times, as well as a trusted source

of randomness consistent across all processes. To this effect, it employs a trusted *Network Time Protocol* (NTP) server to synchronise the clocks of the different nodes, as well as sign timestamps on block headers.

The reputation model assigns scores (in the range of $[0, 1]$) using the following set of rules: 1) processes that send conflicting messages are assigned 0; 2) processes that contribute to an accepted block, by being the proposer of that block or voting in its favour, have their scores incremented and; 3) blocks that do not contribute to consensus, by not proposing a block, not voting, or voting against an accepted block, have their scores reduced. To decrease and increase scores, RBFT multiplies the current score by set factors, x and $1 + y$, respectively. Importantly, the value of the reduction factor, $x \in]0, 1[$, results in a much greater penalisation of validators compared to the increment factor's compensation, $1 + y, y \in]0, 0.03[$. These rules are used every time a block is expected to be produced, regardless of one having been accepted or not. In order to reason about inconsistent messages, all nodes hold a digest vector, which is broadcast at the end of any attempted consensus round; this vector holds the digest of the request within 1) the primary's pre-prepare message or 2) the remaining nodes' prepare messages.

In RBFT leaders are selected probabilistically based on their reputation, processes with higher scores have greater chances of becoming the next primary. As a source of randomness, the protocol uses the latest block's header; which, having been timestamped by the NTP server, cannot be manipulated by a adversarial process. If in a certain round no block is produced, the number of consecutive attempts that have not yielded an accepted block is also a factor in the random number generation.

In regards to the differentiated discourse rights mechanism, processes with higher reputation have a greater weight when forming a quorum. Each process is attributed a discourse right value, based on its reputation, where the sum of every node's discourse rights is 1. Instead of requests being accepted based on regular quorum specifications, they are accepted when the sum of the relevant aggregated messages' discourse rights exceeds $(2f + 1)/N$.

3.3.5 Prosecutor

Prosecutor [18] introduces a BFT protocol focused on providing efficiency and dynamic adaptation to Byzantine attacks, while harnessing Proof-of-Work concepts to limit Byzantine behaviour. Using a scheme similar to the non-BFT Raft [33], Prosecutor divides processes into the roles of *Follower*, *Candidate* and *Leader*, it employs one leader communicating with all other processes (followers). In order to trigger leader elections processes have to transition to the candidate role, there are two timers which control this behaviour. Followers, upon receiving $f + 1$ leader complaints, start a timer which is reset when a value is committed. After becoming candidates, processes start another timer which limits the time a leader election can take, if no leader is elected in that period, the process launches another campaign. These timers hold a random element to prevent multiple honest campaigns from dividing votes and unnecessarily delaying leader selection.

Byzantine processes may continuously try to usurp leadership. In order to prevent this, every election campaign must be preceded by a hash computation, the result of this computation must contain a prefix with an amount of identical characters over a given threshold. This threshold increases with the number of skipped terms and the times a given process attempts a leader election. The hash computation uses as base: a random string denoted as *nonce*, the candidate's ID, the term for which it is campaigning and the hash of a *Proof Window* (PW). The proof window represents the candidate's current log and commit state, ensuring an up-to-date request. In order to vote on an election request, followers must: 1) not have voted before in that term, 2) ensure the candidate is not campaigning for a prior term, 3) confirm the candidate's proof window does not conflict with its view of the log and 4) verify the hash computation's result. If, in a previous leadership, a candidate was able to commit k transactions, followers subtract 1 from its required threshold, where k is an expected minimum number of transactions for a correct leader.

Experimental comparison against PBFT and HotStuff has shown that Prosecutor offers suitable performance while showcasing resilience against Byzantine takeover of the blockchain through its Proof-of-Work mechanism.

3.3.6 PrestigeBFT

PrestigeBFT [34] provides Byzantine consensus powered by a reputation-based view-change protocol. The authors identify the reliance on efficient view-changes in modern BFT algorithms to sustain performance and how

the selection of the wrong leader can greatly diminish performance. Selecting crashed processes will force a wait for the timeout of $f + 1$ processes, while Byzantine leaders can skew the production of blocks in their favour or purposely slow down the blockchain. Additionally, honest validators with non-up-to-date blocks will also reflect poorly on the chain's performance, as missing transactions will have to be obtained before being able to make progress.

Noting how crashed processes affect a static leader schedule, the authors selected an active leader rotation approach, which requires a node to actively apply for the leader role, avoiding the inadvertent selection of a crashed leader that a passive round-robin schedule would make. However, Byzantine processes might try to force elections as often as possible to achieve the leader role and slow down and influence the production of blocks. As such, PrestigeBFT proposes a reputation mechanism that avoids the election of non-responsive validators, hinders dishonest attempts at leader elections and scrutinises leaders for faulty behaviour.

PrestigeBFT's reputation system is centred on an integer value denoted *reputation penalty (rp)*. The value of *rp* is calculated on view-change, where, upon triggering a view-change and campaigning for leadership, a process is penalised with increasing *rp*, after which, the process's past behaviour is used to compensate for the accumulated penalties. Two criteria for compensation are defined: 1) *incremental log responsiveness*, which represents the fraction of transactions which were committed since the last view-change; and 2) *leadership zealousness*, based on the notion that, if an honest process's *rp* increases across multiple views, that increase will be gradual. By extracting a numerical value from these criteria, a compensation is obtained, as a result, the new leader's *rp* will increase, decrease, or remain unchanged according to process behaviour.

During the protocol's execution, validators take one of four roles, *follower*, *redeemer*, *candidate* and *leader*. In regular operation, one process is the leader and the remaining are followers. When a follower suspects a leader failure, it broadcasts a *confirm-failure* request, which awaits $f + 1$ confirmations. When a failure is confirmed, the node responsible for the initial request turns into a redeemer. Redeemers increment their view number and update their *rp*, this is used to determine the difficulty of a hash puzzle; completing this puzzle is exponentially harder the higher a proccess's *rp*, dissuading Byzantine behaviour and affording time for honest processes to become candidates first. Candidates broadcast a campaign message to which followers respond with a vote if the candidate is at least as up-to-date as themselves; in addition to verifying all the steps the candidate took before sending the campaign message were correctly performed, followers cast only one vote in a given view. By acquiring $2f + 1$ votes, a new leader is elected, this leader creates a special *View-Change Block* which holds information pertaining to the leader's election and the reputation system.

In order to avoid conflicting leader elections in a given view, honest nodes employ a random delay when detecting a failure; additionally, when assessing if a candidate is up-to-date, voting nodes take the opportunity to synchronise by fetching more recent blocks from the candidate. To prevent the reputation system from converging with highly penalised honest leaders, when at least $f + 1$ validators surpass a threshold of π *rp*, reputation penalties can be refreshed to their initial values.

PrestigeBFT's consensus protocol is organised similarly to HotStuff, with the leader in the centre of a star pattern. In PrestigeBFT's approach the leader holds only two vote collection rounds, in opposition to HotStuff's three; HotStuff's third round is used to sync honest processes on the committed value, since PrestigeBFT's synchronisation during view-change achieves this function, less communication rounds are needed per commit.

3.3.7 SmartTree

SmartTree [20] was developed in order to enable the dynamic adaptation of the Kauri tree, mitigating downsides identified in the Kauri protocol. It proposes a variant of Kauri in which configurations dynamically adapt to the execution of the protocol in order to respond to faulty processes and non-uniform network latency.

Through SmartLog, a BFT distributed log developed alongside SmartTree, system measurements are shared across all processes. *Latency* between replicas is one of the selected metrics; after a tree fails, the system reverts to the star topology to retake latency measurements between the leader and the remaining nodes without incurring delays from message propagation on a tree. *Misbehaviour* measurements reflect provable infringements of the protocol, such as sending invalid signatures or inconsistent messages to different nodes, revealing faulty processes. *Suspicion* measurements reflect non-provable faults, including crashes, censoring, or network instability; using the latency measurements as basis for the expected time to respond, processes issue complaints to ancestor/successor nodes on the tree.

These measurements are essential to the creation of optimised trees, as faulty nodes should not be used as inner nodes. Latency measurements are, likewise, crucial to ensure high-performing trees. To distribute the load of finding suitable configurations, multiple nodes use simulated annealing [35] to non-deterministically generate trees. SmartLog is used to disseminate the generated trees, which are able to be deterministically chosen at reconfiguration time through the use of heuristics.

Using these techniques, SmartTree enables the selection of more optimised trees than the ones from Kauri. Additionally, SmartLog is able to recognise when the current tree's performance has been compromised though changes in the environment and pre-empt a view-change.

3.4 Discussion

Table 1 contains a brief comparison of key features found in the aforementioned state-of-the-art algorithms. Next, we discuss each feature in regards to the various state-of-the-art adaptive solutions and how they relate to our work and intended approach.

Protocol	Topology	Byzantine behaviour-aware	Epoch-based	Chain quality
Carousel [17]	Star	✗	✗	✓
Hammerhead [29]	Clique	✗	✓	N/A†
ABSE [32]	Clique/ Star/ others	✗	✗	N/A‡
RBFT [19]	Clique	✓	✗	✓
Prosecutor [18]	Star	✓	✗	✗
PrestigeBFT [34]	Star	✓	✗	✗
SmartTree [20]	Tree	✓	✗	N/A‡
Our Approach	**Tree**	✓	✓	✓

† The Hammerhead protocol does not concern itself with chain quality as it's inherent to DAG-based protocols.
‡ ABSE and SmartTree leave selection criteria to implementation, chain quality concerns depend on implementation.

Table 1: Comparison of key features of our approach with existing adaptive BFT solutions

Topology Systems that aim at reconfigurating protocols that use clique or star communication topologies focus on identifying which processes are better candidates to fulfil the leader role. For this purpose, they need to rank individual processes, which is simpler than ranking trees, given that the number of potential trees is factorial with the number of processes. Therefore, systems concerned with the reconfiguration of tree-based protocols need to address the problem of searching for suitable trees, a task that may be computationally expensive. Also, the range of roles (root, inner node, leaf node) is wider with makes harder to assess the behaviour of nodes and assign suitable reputation values.

Byzantine behaviour-aware We aim at devising an adaptation strategy for tree-based blockchain protocols where Byzantine behaviour needs to be addressed. Thus, the adaptation mechanisms must be robust to adversarial manipulation attempts. Algorithms like Carrousel and Hammerhead, while applied over BFT consensus, confine their adaptation schemes to a crash-fault assumption. By not taking into account Byzantine faults, these algorithms may fall victim to adversarial processes which exploit those reputation schemes to behave maliciously while undetected. RBFT and SmartTree employ strategies to capture equivocating nodes, capturing one possible Byzantine behaviour. Additionally, SmartTree aims to capture censoring and omission of messages through a report system narrowing faulty behaviour to a pair of nodes for each reported infringement of protocol. Finally, Prosecutor and PrestigeBFT hold nodes suspect of abusing their Raft-like election mechanisms to constraints which hinder future election attempts, enabling correct processes to operate regularly and hold fair elections when truly necessary.

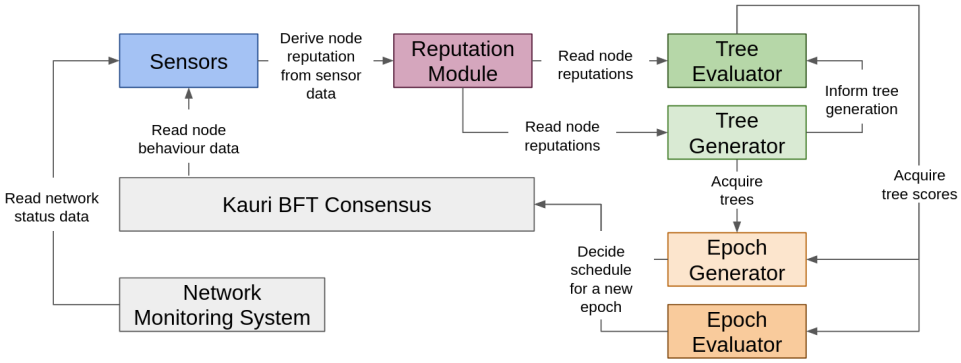

Figure 1: Proposed architecture

Epoch based Among these state-of-the-art algorithms, Hammerhead presents the concept of epochs. Albeit it has been proposed in the context of DAG-based consensus, we find the notion of epoch to be useful for the adaptation of trees. By defining a period of execution from which relevant information is extracted to inform the reputation scores, we increase the amount of data available to inform the reputation decision, especially by analysing different tree configurations; as well as mitigate possible imprecisions due to exceptional behaviour, such as latency spikes. Another advantage of dividing a protocol's execution into epochs is the concentration of decisions in well defined points. Taking into account the generation of trees might pose a relevant computational effort, epochs lend time for a node, or set of nodes, to generate multiple trees before making the best decision on the next epoch's schedule.

Chain quality As stated above, the property of chain quality (along with the strongly linked fairness) is a desirable property in blockchain systems. Most of the explored systems were either directly concerned with chain quality, or left details up to implementation where chain quality could become one of the concerns. Conversely, Prosecutor and PrestigeBFT propose schemes where a single process could hold the leader slot indefinitely. When coordinating leader decisions it is important to assure a fair representation of correct leaders while minimising the representation of Byzantine processes.

As explained above, Byzantine behaviour awareness and chain quality are important properties to achieve in an adaptive BFT system. Similarly, we find the concept of epochs will aid us in our goal of adapting tree-based blockchains. In the following section, we will show how we intend to approach building an epoch-based adaptive BFT system, harnessing the strengths of the tree pattern and providing Byzantine behaviour awareness and chain quality.

4 Approach

We now present our architecture to support the dynamic reconfiguration of tree-based Byzantine consensus protocols for the blockchains. Our proposal is to use an *epoch based* strategy, where a sequence of target trees is scheduled to be used during the next period of operation, based on performance and reputation information collected from previous epochs. To select a suitable sequence of trees we will define metrics to rank a schedule which, in turn, will be based on metrics to rank each individual tree that composes the schedule. The rank of a tree will be based on both network performance parameters (latency, CPU utilisation, etc) and and past compliance with the protocol specification of individual nodes, that will be captured by a *reputation score*. The system will use a MAPE-K control loop [36] that will use sensors to collect performance data, a planning component to derive future schedules, and actuators that ensure that all correct processes adopt the same appropriate schedule. The architecture is depicted in Figure 1. In the remainder of this section we discuss the rationale behind our design choices, describe the operation of each component, discuss strategies to optimise the system, and present directions for further improvement.

4.1 BFT Consensus Module and System Model

We will use the Kauri BFT Consensus protocol [15], along with the same system model assumptions. We consider a system composed of a set of processes $\Pi = \{p_1, p_2, ..., p_N\}$ subject to Byzantine faults, where at most $f < \frac{N}{3}$ processes may display faulty behaviour. We assume a partially synchronous network where messages can be arbitrarily delayed until GST, after which there is a known Δ bounding message latency.

4.2 Sensors

Sensors enable the extraction of information from the blockchain execution. We install these sensors on the system nodes, where they can collect internal metrics (e.g. CPU utilisation) or capture information from the established links (e.g. network latency). From the available data sensors can extract, we define two types of indicators: network performance indicators and node behaviour indicators.

Blockchains are based on a decentralised operation. Ideally, this decentralisation entails global dispersion of nodes with a variety of hardware components, software components, and operating conditions [37]; this results in heterogeneous networks. Configurations adapted to the network topology can provide better performance [26]; we aim to capture information from the network and construct a notion of node distances in the network. As such, we employ network sensors to carry out measurements of indicators like throughput, latency, jitter and CPU utilisation.

As discussed above, Byzantine behaviour-aware algorithms can better counter malicious processes and guarantee reliable progress. Node behaviour sensors intend to capture the actions of nodes in order to assess their compliance to protocol and, in some cases, collect evidence that some process is Byzantine. To accomplish this, node behaviour sensors will report activity such as block endorsement, sharing of inconsistent messages, and message logs.

Considering that sensors will execute in the existing nodes, it is necessary to assume that sensors may be also subject to Byzantine faults. As a result, our approach must be able to deal with incorrect measurements. When a value is reported by at least $f+1$ nodes, we can assess its correctness, this is because at least one honest sensor is reporting that value; if not enough samples are able to be taken (e.g. link latency), the system can employ heuristics to extract from conflicting measurements which is most likely correct, or, when applicable, require cryptographic evidence for the reported data.

4.3 Reputation Model

The reputation module aims at computing a score value that captures the node behaviour. Our reputation mechanism seeks to reward honest behaviour and penalise Byzantine behaviour in order to promote the construction of better configurations. Due to the nature of the tree topology, node connections are kept minimal; which, while critical for performance, makes it harder to gather precise knowledge about the system, especially under Byzantine faults. In order for any given node to obtain a global view of the system behaviour, the distributed sensors must share their measurements.

We model the reputation score as an estimated probability of a node to display correct behaviour, with score values ranging in the $s \in [0, 1]$ interval. Modelling reputation such that its possible values are well defined and bounded is useful for the purposes of ranking configurations, as detailed in Section 4.4.2. Considering N nodes where at most $f = \frac{N-1}{3}$ nodes are faulty, we may attribute an initial score of $s_i(0) = \frac{2}{3}$ to each node p_i and define a set of rules to reward/penalise behaviour observed from sensors:

$$s_i(t) = 0, \text{ if } p_i \text{ displays provable Byzantine behaviour} \tag{1}$$

$$s_i(t) = k_1 \times s_i(t-1),\ 0 < k_1 < 1 \text{ if } p_i \text{ belongs to a faulty link} \tag{2}$$

$$s_i(t) = k_2 \times s_i(t-1),\ 0 < k_2 < 1 \text{ if the leader, } p_i, \text{ is replaced ahead of expected} \tag{3}$$

$$s_i(t) = k_3 \times s_i(t-1),\ k_3 > 1 \text{ if } p_i \text{ endorses a committed block} \tag{4}$$

Through simulations the values of k_1, k_2, k_3 can be fit for best performance, defining how the system perceives deniable Byzantine behaviour, how harshly it is punished and how its absence is compensated. With regard to Rule 1, which handles provable Byzantine behaviour, the system should consider the node to be temporarily

compromised, halting score adjustments until a set timeout expires or enough epochs elapse. Furthermore, Rule 2 penalises nodes in a "faulty link". These links are treated as faulty if it, or one of its nodes, behaves such that it hinders timely communication within the tree; Section 5.1.3 elaborates on how we intend to measure and extract these faults.

4.4 Tree Generation and Ranking

In order to create schedules we must first tackle the individual configurations that form those schedules, in tree-based systems, the tree constitutes the individual configurations.

4.4.1 Generation

As discussed above, the generation of optimal trees may be computationally expensive. We consider two approaches for the tree generation algorithm, namely: deterministic or non-deterministic. The advantage of a deterministic algorithm lies in its reproducibility across any node containing the same view of the system, as well as enabling the creation of trees in consistent time, compared to the non-deterministic alternative, which may arrive at solutions at inconsistently faster or slower speeds. Conversely, the creation of an algorithm which provides n trees while under time complexity constraints might pose a significant challenge. The non-deterministic approach sacrifices time and computational resource certainty to provide wider exploration of the result space. We plan to use a deterministic algorithm, as the prospects provided by this solution will enable more direct approaches to the decentralisation of the trusted controller, given we verify satisfactory results from our proposed algorithm.

To generate a tree we make use of Kauri's [15] concept of bins, dividing the validators into groups that guarantee at least $m - f$ bins free from faulty processes, with m representing the number of bins and f the number of faulty nodes. By sorting the validators by their reputation scores we may divide them into bins of ascending reputation. The bins containing the most reputable nodes are the most likely not to contain any Byzantine processes. Therefore, by using one of those bins to fill the tree's internal nodes we create a robust configuration with high probability. By rotating the internal nodes of the tree we can easily create more likely desirable configurations.

We believe this algorithm would be able to provide a sufficient amount of configurations for the purposes of creating a schedule, while avoiding seeming shortcomings of the strategy. 1) The algorithm takes into account only the behavioural data of the nodes to create configurations, one could expect the presence of unresponsive trees as a result. We argue that our reputation scores are not an independent variable from the nodes' network conditions; for instance, a slow node, unlikely to respond in time to be used to form a quorum, will have its reputation negatively affected; as such, it would also be pushed to the leaves of the tree through this algorithm, which is beneficial. 2) Due to the nature of the topology, a slow internal node might harm the reputation of its descendents. While at first this is the case, when recalculating the bins after this hit to scores, the slow node is likely moved to the leaves of the tree along with the nodes which were previously dependent on it. As they then occupy the same layer of the tree, the harmed nodes are now independent of the slow node and may regain reputation. In short, in spite of immediate negative effects to node reputations, in the long term, scores should converge to depict reality regardless of these events.

4.4.2 Ranking

To gauge the expected performance of a tree, nodes must be able to evaluate them. The assessment of the quality of given configuration must use an efficient algorithm, as it is expected that, in a trustless environment, nodes will process large numbers of trees. As our nodes possess separate network and behaviour data, the ranking will consider these aspects individually, before unifying them into a single value.

Network Scoring Regarding latency, we consider a good tree one which delivers the leader's message to the necessary amount of nodes in the least possible time. In a blockchain system, there are alternative trees which can propagate the same message from the root to a leaf in the same time. However, these trees may incur different delays in their upper and lower links. We consider a tree which sets slower links closer to the leaves to

be better. Although, to gather a quorum of votes, one leaf node may be more desirable for its fast connection to the root; we may need to receive a vote from a slower leaf, as the faster one is render unavailable by Byzantine action. As such, faster links at the top of the tree mean less latency to arrive at these other leaves. We propose to evaluate a tree through the minimum cumulative sum of the latency for a message to arrive from the root node to a Byzantine quorum of nodes. The cumulative sum will account the latency of edges closer to the root more times, which favours lower latency at the top of the tree.

$$\min_{\substack{Q \subseteq \Pi \\ |Q|=2f+1}} \left(\sum_{p \in Q} NetworkDistance(root \rightarrow p) \right)$$

Behaviour Scoring Since the reputation scores represent the probability of a node's correct behaviour, we consider a good tree positions nodes so that messages have a high probability of arriving at every node. The probability of a given node receiving a message depends on its ancestor nodes and results from the probability of all of them behaving correctly. Considering $P(p_x \rightarrow p_y)$, the probability of all nodes from p_x (inclusive) to p_y (exclusive) behaving correctly, we define the behaviour ranking of a tree as given by:

$$\sum_{i=0}^{N} P(root \rightarrow p_i)$$

Unified Score To compare trees in an efficient manner, we propose combining the prior rankings into a single number. This entails the application of some dimensionality reduction technique, ranging from normalising each measurement and performing a weighted average, to the usage of *Space Filling Curves*, such as Hilbert Curves, to reduce multiple types of data to a single dimension [38, 39].

The usage of these techniques takes advantage of bounded spaces. Since our reputation model is defined as bounded, it is appropriate for this use. Although at first the network measurements may not seem suitable, as they are not bounded; through the estimation of ideal network conditions we can also bound them.

4.5 Epochs

As discussed previously, epochs provide discrete slots wherein the system is able to provide well-defined schedules and set clear transition points to separate sensor data for the calculation of these schedules. The mentioned aspects favour a fluid execution of the protocol, while minimising the effects of asynchrony and lending time for the performance of more complex computations epoch scheduling may require.

Our system establishes a three-epoch sequence for the creation and implementation of a given epoch schedule. Providing the protocol is preparing the schedule for epoch E, we 1) take the readings gathered from the execution of epoch $E-2$ to base our view of system, 2) in epoch $E-1$, sensor readings arrive at the decision modules, triggering reputation, tree and schedule building, 3) in epoch E, the schedule is put in practice. These tasks are pipelined as shown in Figure 2, ensuring the continuous and timely delivery of epochs to the BFT consensus module.

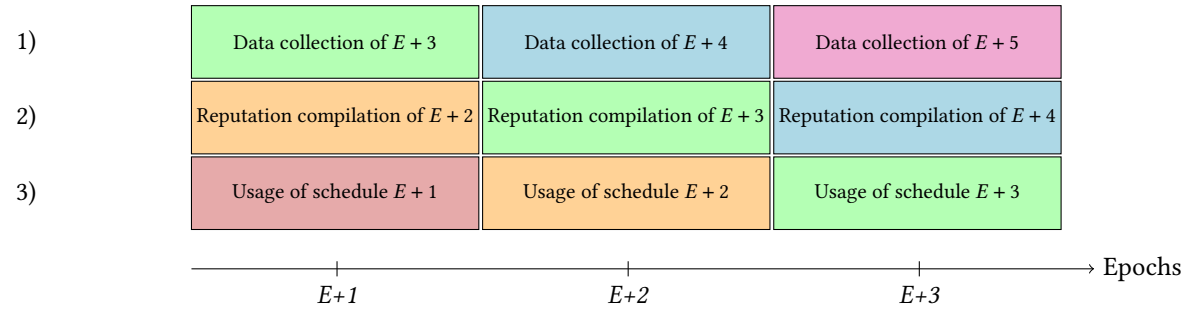

Figure 2: Pipelining of the tasks to create a schedule.

Due to the nature of our epoch scheme, there must be at least two pre-established epochs for the system to collect data and reason about before generating dynamic schedules. We propose using these epochs to fairly rotate across randomly instantiated bins, allowing nodes to occupy varying locations in the configurations. This would avoid biasing reputations to any process or bin and grant the opportunity to properly reward/punish node behaviour.

4.5.1 Schedule Generation

Having described how sensor data is used to form trees, we now focus on how each epoch is assigned a schedule and how that schedule is created. When forming a schedule there are two aspects we need to take into account: 1) schedules must contain the best trees according to our Tree Evaluator Module and 2) schedules must follow principles of fairness and chain quality.

We propose a greedy algorithm, detailed in Algorithm 1, to determine a schedule S by regarding the available trees produced by the Tree Generator Module, $\mathcal{T}$, taking the highest rated trees and adding them to S. Importantly, to satisfy our fairness and chain quality concerns, we scale the tree scores by a factor $\lambda(\alpha)$, where λ is a function that decreases with the amount of times, α, that the tree's leader is represented in S and M, where M holds the last m previously adopted configurations from the prior epoch(s).

Algorithm 1 Schedule Generation

λ ▷ Scaling function
rootCount ▷ Counts how many times the root of a given tree is the root in a set of trees
$S \leftarrow \emptyset$
while $|S| \neq$ SCHEDULE_SIZE **do**
  $T \leftarrow \arg\max_{\tau \in \mathcal{T} \setminus S} (\tau.score \cdot \lambda(rootCount(\tau, S \cup M)))$
  $S \leftarrow S \cup \{T\}$
end while
return S

4.5.2 Schedule Scoring

The score of a schedule will be a function of the individual scores of the trees that compose the schedule. We consider different alternatives functions to derive the score such as SUM, MIN, AVERAGE, MEDIAN, etc. We plan to assess the suitability of these scoring functions during the evaluation.

5 Implementation

In this section, we first describe how we plan to implement a prototype of our system and later discuss some potential extensions that we would like to implement if time permits.

5.1 Prototype

Due to the time constraints of this work, we will simplify the implementation of some components. In particular, we will make a first implementation or the architecture where we will a centralised controller. This simplification will ease the implementation of our system, without compromising architecture fidelity or the validity of our findings.

Additionally, given the breadth of components in our architecture, we will try to adapt existing solutions to our implementation; namely, in the modules regarding network monitoring and BFT consensus. We aim to conduct network monitoring through a network coordinate system such as Vivaldi [40], and use an available implementation of Kauri [15] to handle BFT consensus, which we plan to modify in order to integrate it with the remaining modules. This will allow us to focus on the implementation of the reputation model and schedule

generation. We now describe the plans for an initial prototype applying these principles and discuss key aspects to consider in future extensions.

5.1.1 Deployment

The system is based on N server processes, each running an instance of the BFT consensus protocol. Every node is equipped with network sensors, to capture information on the server itself or its connections, and node behaviour sensors, which report actions observed from the consensus protocol's execution. In our implementation, we will use a reduced number of measured attributes. We find this focused approach will lead to a broad enough understanding of system status to drive adaptation, while still leaving room to tune the amount of metrics.

5.1.2 Network Sensors

Our objective in terms of network measurements is to build a data structure capturing the distance between nodes in our network, such as a latency matrix [20,41] or network coordinates [40,42]. To create network matrices, each node creates a vector containing the latency to every other node, these vectors are combined to form a matrix comprising every possible link. There are several approaches to the measurement of latency, from raw network latency, which can be measured through simple techniques, like pinging; to protocol latency, which includes the delay incurred from processing data from different steps in the algorithm, and is usually measured by timing the delay of existing protocol messages. Disregarding noise, network matrices are symmetrical; however, even when dealing with two honest nodes, their sensors may report different, yet possibly similar, measurements. Since Byzantine nodes can try to distort the real latency they face when communicating with a certain node, many protocols sanitise their network matrices by trusting the higher value measurements to form a symmetrical matrix. Network coordinates make use of these latency measurements to represent nodes in a Euclidean space where node distances approximate the observed latencies. Network coordinates allow for fast integration of new nodes into the coordinate system and have been shown to work under Byzantine faults [43].

5.1.3 Node Behaviour Sensors

By monitoring node behaviour, we are trying to determine which are more likely to commit faults. When defining Byzantine behaviour, it is useful to divide it into two categories, similarly to the ones in SmartTree [20]. Some node behaviour can be irrefutably linked to Byzantine behaviour, such as equivocation; where incompatible messages are sent and signed by the same process and holding these messages enables the conclusion that the sender is infringing the protocol. In contrast to this conduct, which we call *provable Byzantine behaviour*, we define *deniable Byzantine behaviour* as that which can be explained by external factors such as network asynchrony. A node which does not receive a message cannot distinguish this from a purposeful infraction or uncontrollable network instability; thus, one could deny their Byzantine behaviour when performing such actions. Due to the nature of these types of behaviour, the means to extract them are to be different themselves.

Provable Byzantine Behaviour To assess provable Byzantine behaviour we focus on equivocation. To prevent nodes from sending conflicting messages to different processes, the children of a given node should exchange an authenticated digest the parent's messages in every round. While the absence of a message would fall within deniable Byzantine behaviour, two conflicting messages prove the node is faulty.

Deniable Byzantine Behaviour In order to capture deniable Byzantine behaviour we elect to monitor 1) leaders which have not fulfilled their terms to the end and 2) block endorsers. A leader has influence over the whole tree, if the underlying nodes choose to depose it before the allotted time, it means the leader is purposefully withholding progress, or the configuration is compromised. By collecting the signatures in a block we can identify faulty links (i.e. links containing a faulty node). In trees, nodes will only be able to manipulate their whole underlying sub-trees, as message dissemination cannot bypass nodes and threshold cryptography prevents granular manipulation of votes; thus, the maximal sub-trees excluded from consensus display a fault at the links above their roots. From a faulty link, we can conclude one of its nodes is faulty (or experiencing temporary network constraints, which is undistinguishable to the sensor). Additionally, since, as mentioned,

aggregated votes are inseparable, the leader is likely to include more votes than strictly necessary to form a quorum certificate; this reassures honest participants will not be suspected of being Byzantine unnecessarily.

5.1.4 Controller

As hinted before, we will start by implementing a centralised controller. We assume that this component possesses sufficient computational power and is connected to the network in a manner that it is equipped to handle all assigned computations and data transfers, while incurring negligible drops in performance from the centralised load. The centralised controller acts as a trusted component to all correct processes. It acquires readings from the distributed sensors through an appropriate means. By possessing the sensor readings, it can calculate node reputations and perform the computations to derive tree configurations and schedules. The controller is also tasked with informing every process of the decided epoch schedules, guaranteeing a smooth execution of the BFT protocol.

5.2 Extending the Implementation

We now focus on the further development of our solution. In order to better suit our approach for a real-world scenario, there are several aspects we must explore, namely: 1) selecting a wider array of features to inform schedule creation, 2) decentralising the schedule decisions and 3) exploring alternatives for distributed schedule generation, if justified by generation efficiency and load.

5.2.1 Selection of Features

Our approach to the implementation of sensors allows for a finer tuning of the selected data features to measure. In order to optimise the results we obtain when reasoning on the collected data, sensors may be installed to collect data on additional features. Network sensors can capture connection bandwidth, throughput, as well as node processing power, which affect network performance under the protocol's load. To better encompass node behaviour, we can introduce a complaint system, where nodes report when their peers do not behave correctly to a shared log, or introduce a more extensive logging solution which does not omit seemingly correct behaviour. These logs provide direct information on both provable and deniable Byzantine behaviour; furthermore, their introduction enables the expression of additional provable Byzantine behaviour, when nodes log entries incompatible with the observed reality.

5.2.2 Decentralisation of Decisions

A centralised controller ensures a shared view of the system when conducting decisions. When moving away from that simplification, it is important to consider that, due to asynchrony and Byzantine behaviour, most nodes will not share the same view; as such, a given degree of central decision is necessary to ensure enough nodes adopt the same configuration. To achieve this we propose a BFT approach inspired by the Raft [33] algorithm, similar to Prosecutor [18] and PrestigeBFT [34]. Using this approach, a candidate initiates an election by presenting an epoch's schedule and, given a sufficiently well ranked schedule, enough of remaining the nodes will converge on that schedule, in spite of view disagreements.

5.2.3 Alternatives for Schedule Generation

As stressed before, searching for configurations is a more complex problem when dealing with trees than with other studied patterns. In Section 6 we detail the methods we will use to evaluate the efficacy of our scheduling strategy. In the absence of a deterministic algorithm capable of finding an array of suitable configurations, we consider the exploration of non-deterministic searching algorithms. Parallelly, considering the possible costs of running these algorithms independently in every node, we suggest examining a cooperative solution to tree generation, where processes contribute to a single configuration pool. Furthermore, to avoid bad faith cooperation, a cooperative solution must contain incentives for nodes to contribute with configurations that promote a performant schedule, rather than their own interests.

6 Evaluation

This section is centred on how we intend to evaluate the results of our work, showing its contributions and how it compares to the state-of-the-art.

To accurately evaluate a distributed system, we must provide an appropriate testing environment, reflective of real-world deployments. This requires nodes with adequate processing power and an heterogeneous network, which can be achieved through a cluster of distributed machines and network emulation [44].

We will compare performance metrics of our solution with the state-of-the-art, making use of consensus latency and throughput measurements across deployments of varying size and network conditions.

We plan to evaluate the performance of our architecture in faultless conditions. First, we will deploy the system in a static network, both in homogeneous and heterogeneous network conditions. Second, we will evaluate how the system adapts to dynamic network events, such as link removals or latency spikes.

We aim to gradually introduce faulty nodes to the network until we reach the limit defined by the system model. We intend to assess how the protocol handles different proportions of Byzantine processes and how they affect performance metrics relative to the state of the art. Byzantine nodes dispose of several attack options to compromise the protocol, including:

- *Censoring & Omission*, where faulty processes exclude information from/ refrain from propagating messages to correct servers;
- *Equivocation*, where faulty processes send conflicting messages to different servers;
- *Timeout attacks*, where faulty processes match suspicion timeouts with correct servers;
- *Quiet participation*, where faulty processes do not respond to any request;
- *Slowdown*, where faulty processes delay their messages within the protocol's timeouts;
- *Collusion*, where faulty processes cooperate to hinder the protocol.

We consider several attack objectives that guide Byzantine actions and define the strategies they may use. These objectives entail:

- *Targeted disenfranchisement*, Byzantine nodes select a target or group of targets and act to reduce their participation in the protocol. Adversarial nodes may define individual targets or collude to attack the same set of victims. Notably, Byzantine nodes will behave correctly when they cannot affect target nodes;
- *Leadership seizure*, attackers will try to obtain positions of influence (root and inner nodes). Through periods of good behaviour, Byzantine processes garner reputation in order to facilitate the adoption more favourable configurations where they can control the production of blocks;
- *Performance deterioration*, Byzantine servers attempt to slow down the whole blockchain system as an end in itself. As such, they work to prevent the aggregation of a Byzantine quorum of votes by the leader;
- *Aimless faults*, we consider that in some cases processes may display Byzantine behaviour haphazardly.

To better evaluate the precision of our reputation model and scheduling algorithm, we will extract how the system ranks the different nodes during execution and how the selected configurations place them while subject to the various attack strategies. We will analyse how the ranks of faulty nodes compare to the ranks of correct nodes and how faulty nodes integrate the generated trees, regarding their proportion within inner and leaf nodes.

Lastly, considering how we structured our first implementation, we have to consider how the simplifications we made affect the obtained results. Since, functionally, it accurately represents the intent of the architecture, aspects, such as the identification of faulty nodes and the efficacy of generated schedules, are unaffected. However, performance indicators will lessen impacts or overestimate gains, as the centralised controller absorbs a portion of the solution's overhead.

7 Work Schedule

The Gantt chart in Figure 3 illustrates the planned schedule for various tasks related to the MSc dissertation.

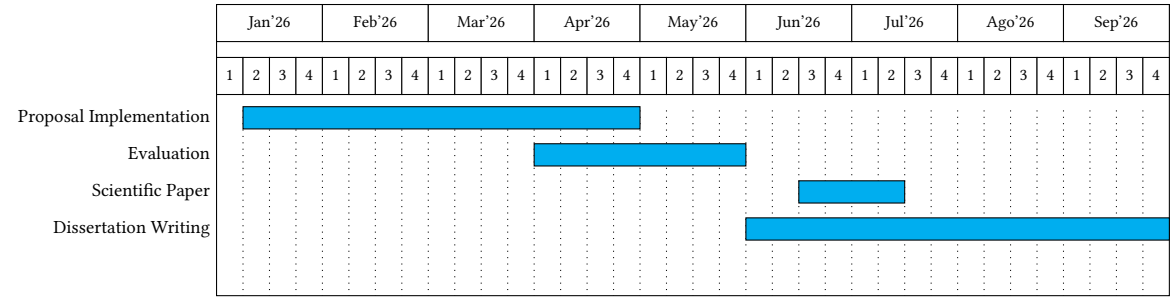

Figure 3: Planned Schedule

As portrayed above, the scheduled work entails: 1) implementing the proposed architecture, 2) designing tests and performing the evaluation of the implemented solution, 3) writing a paper describing the project, and 4) writing the dissertation and preparing its presentation.

8 Conclusions

Blockchains provide important functionality for the development of distributed applications in trustless environments. Permissioned blockchains make use of BFT consensus protocols which grant finality to the blockchain's decisions. As a central component to these blockchains' performance, consensus protocols need to provide reliable and timely progress to ensure the well functioning of the blockchains and their underlying applications. The dynamic adaptation of BFT protocols can enhance their reliability and provide continuous performance under adversarial pressure.

This work focuses on the dynamic adaptation of tree-based blockchain protocols. While the tree communication pattern has shown promising results to improve the performance of BFT protocols, no practical solution for the adaptation of tree based protocols has been found. In developing our architecture, we aim to provide a reputation model and scheduling solution that captures Byzantine behaviour to generate efficient trees that do not compromise on fairness and chain quality. By dividing the protocols into discrete epochs, we intend to optimise the collection and transformation of protocol data into practical sequences of resilient configurations.

This report compiles the state-of-the-art analysis we conducted to inform the creation of our architecture, showcases the approach and design decisions taken when developing an architecture capable of supporting the adaptation of trees. Finally, this document presents our considerations for the implementation and expansion of our work, as well a methodology to evaluate our proposal.

Acknowledgment. This work was supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 and UID/PRR/50021/2025 and GLOG (financed by the OE with ref. LISBOA2030-FEDER-00771200).

Bibliography

[1] P. Treleaven, R. G. Brown, and D. Yang, "Blockchain technology in finance," *Computer*, vol. 50, no. 9, pp. 14–17, 2017.

[2] J. R. Varma, "Blockchain in finance," *Vikalpa*, vol. 44, no. 1, pp. 1–11, 2019.

[3] V. P. Ranganthan, R. Dantu, A. Paul, P. Mears, and K. Morozov, "A decentralized marketplace application on the ethereum blockchain," in *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*. IEEE, 2018, pp. 90–97.

[4] C. Liu and Z. Li, "Comparison of centralized and peer-to-peer decentralized market designs for community markets," *IEEE Transactions on Industry Applications*, vol. 58, no. 1, pp. 67–77, 2022.

[5] L. Preto, S. Eisa, O. Remédios, D. R. Matos, and M. L. Pardal, "Dependable food traceability data through blockchain and database integration," in *2025 20th European Dependable Computing Conference Companion Proceedings (EDCC-C)*, 2025, pp. 163–167.

[6] M. M. Queiroz, R. Telles, and S. H. Bonilla, "Blockchain and supply chain management integration: a systematic review of the literature," *Supply chain management: An international journal*, vol. 25, no. 2, pp. 241–254, 2020.

[7] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized business review*, 2008.

[8] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: the works of leslie lamport*, 2019, pp. 203–226.

[9] T. Swanson, "Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems," *Report, available online*, 2015.

[10] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for wans," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 369–384.

[11] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 358–372.

[12] A. Turcu, S. Peluso, R. Palmieri, and B. Ravindran, "Be general and don't give up consistency in geo-replicated transactional systems," in *International Conference on Principles of Distributed Systems*. Springer, 2014, pp. 33–48.

[13] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 99, no. 1999, 1999, pp. 173–186.

[14] 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*, 2019, pp. 347–356.

[15] 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.

[16] R. Pass and E. Shi, "Fruitchains: A fair blockchain," in *Proceedings of the ACM symposium on principles of distributed computing*, 2017, pp. 315–324.

[17] S. Cohen, R. Gelashvili, L. K. Kogias, Z. Li, D. Malkhi, A. Sonnino, and A. Spiegelman, "Be aware of your leaders," in *International Conference on Financial Cryptography and Data Security*. Springer, 2022, pp. 279–295.

[18] 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.

[19] K. Lei, Q. Zhang, L. Xu, and Z. Qi, "Reputation-based byzantine fault-tolerance for consortium blockchain," in *2018 IEEE 24th international conference on parallel and distributed systems (ICPADS)*. IEEE, 2018, pp. 604–611.

[20] H. Gogada, C. Berger, L. Jehl, H. P. Reiser, and H. Meling, "Smartlog: Metrics-driven role assignment for byzantine fault-tolerant protocols," *arXiv preprint arXiv:2502.15428*, 2025.

[21] J. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 281–310.

[22] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," https://ethereum.org/whitepaper/, 2014.

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

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

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

[26] H. Teixeira, "Self-adapting bft consensus: Leveraging heterogeneity in dissemination/aggregation trees." Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, 2023.

[27] T. Pereira, "Dynamic trees for byzantine consensus protocols," Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, 2024.

[28] G. Carvalho, "Leader selection in byzantine consensus protocols," PIC2 - Master in Computer Science and Engineering, Instituto Superior Técnico, Universidade de Lisboa, 2025.

[29] G. Tsimos, A. Kichidis, A. Sonnino, and L. Kokoris-Kogias, "Hammerhead: Leader reputation for dynamic scheduling," *arXiv preprint arXiv:2309.12713*, 2023.

[30] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is dag," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 165–175.

[31] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: Dag bft protocols made practical," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2705–2718.

[32] X. Liu, Z. Zhang, Z. Li, H. Yin, M. Li, J. Liu, M. Conti, and L. Zhu, "Abse: Adaptive baseline score-based election for leader-based bft systems," *IEEE Transactions on Parallel and Distributed Systems*, 2025.

[33] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.

[34] G. Zhang, F. Pan, S. Tijanic, and H.-A. Jacobsen, "Prestigebft: Revolutionizing view changes in bft consensus algorithms with reputation mechanisms," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 1930–1943.

[35] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[36] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[37] C. Ovezik, D. Karakostas, and A. Kiayias, "Sok: A stratified approach to blockchain decentralization," in *International Conference on Financial Cryptography and Data Security*. Springer, 2024, pp. 128–155.

[38] H. Sagan, *Space-filling curves*. Springer Science & Business Media, 2012.

[39] H. V. Jagadish, "Linear clustering of objects with multiple attributes," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, 1990, pp. 332–342.

[40] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris, "Practical, distributed network coordinates," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 1, pp. 113–118, 2004.

[41] C. Berger, H. P. Reiser, J. Sousa, and A. Bessani, "Aware: Adaptive wide-area replication for fast and resilient byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1605–1620, 2022.

[42] P. Pietzuch, J. Ledlie, M. Mitzenmacher, and M. Seltzer, "Network-aware overlays with network coordinates," in *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06)*, 2006, pp. 12–12.

[43] Y. Zhou, "Computing network coordinates in the presence of byzantine faults," Master's thesis, Massachusetts Institute of Technology, 2009.

[44] S. Amaro, M. Matos, and V. Schiavoni, "Kollaps: Decentralized and efficient network emulation for large-scale systems," *IEEE Transactions on Networking*, vol. 33, no. 1, pp. 35–50, 2025.