Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation

Ray Neiheiser
INESC-ID, IST, U. Lisboa & UFSC/DAS
Portugal & Brazil

Miguel Matos
INESC-ID, IST, U. Lisboa
Portugal

Luís Rodrigues
INESC-ID, IST, U. Lisboa
Portugal

Abstract

With the growing commercial interest in blockchains, permissioned implementations have received increasing attention. Unfortunately, the BFT consensus algorithms that are the backbone of most of these blockchains scale poorly and offer limited throughput. Many state-of-the-art algorithms require a single leader process to receive and validate votes from a quorum of processes and then broadcast the result, which is inherently non-scalable. Recent approaches avoid this bottleneck by using dissemination/aggregation trees to propagate values and collect and validate votes. However, the use of trees increases the round latency, which ultimately limits the throughput for deeper trees. In this paper we propose Kauri, a BFT communication abstraction that can sustain high throughput as the system size grows, leveraging a novel pipelining technique to perform scalable dissemination and aggregation on trees. Our evaluation shows that Kauri outperforms the throughput of state-of-the-art permissioned blockchain protocols, such as HotStuff, by up to 28x. Interestingly, in many scenarios, the parallelization provided by Kauri can also decrease the latency.

CCS Concepts: • **Computer systems organization → Reliability**; **Fault-tolerant network topologies**.

Keywords: Distributed Systems, Fault Tolerance, Blockchain

1 Introduction and Related Work

The increasing popularity of blockchains in addressing an expanding set of use cases, from enterprise to governmental applications [9], led to a growing interest in permissioned blockchains, such as Hyperledger Fabric [5]. In contrast to their permissionless counterparts, permissioned blockchains can ensure deterministic transaction finality, which is a key requirement in many settings [31], and can offer high throughput in small sized systems [33].

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483584

However, emerging use cases for permissioned blockchains require the system to scale to hundreds of participants [16]. For instance, the Diem blockchain states that: "Our goal was to choose a protocol that would initially support at least 100 validators and would be able to evolve over time to support 500–1,000 validators" [13]. In addition to that, a recent paper from IBM [28] discusses the need to extend HyperLedger to support deployments above 100 nodes in order to address the requirements of platforms such as Corda [26]. However, most permissioned blockchains are based on variants of classical byzantine fault-tolerant (BFT) consensus protocols that scale poorly with the number of participants [10, 22].

Such scalability limitations stem from bottlenecks both at the network and processing levels that result from the large number of messages that need to be sent, received and processed to reach consensus. For instance, the well-known PBFT protocol [7] organizes participants in a clique and uses an all-to-all communication pattern that incurs in a quadratic message complexity. Although there have been many proposals to extend and improve several aspects of PBFT (such as [3, 11, 29, 30]), most preserve its communication pattern.

In HotStuff [33], only the leader sends/collects messages directly to/from all other processes, i.e. communication is based on a star topology centered at the leader. This approach results in linear message complexity but the leader is still required to receive and validate votes from at least $2f + 1$ processes. At the time of this writing, the publicly available implementation of HotStuff uses *secp256k1* [32], a highly efficient elliptic curve algorithm that is also used in Bitcoin[2]. In this implementation, the leader has to relay the full set of signatures to all processes. Alternatively, it is possible to use multisignatures, such as BLS [4], to reduce the message size at the expense of additional computational load at the leader. However, due to the centralized control, HotStuff is inherently non-scalable: the system performance is limited by the computing and bandwidth capacity of the leader process.

One possible way to circumvent the scalability constraints is to select a small committee such as in Algorand [15] or SCP [24]. However, this approach either reduces the resilience of the system (the maximum number of faults becomes a function of the committee size and not of the entire system size) or compromises deterministic finality (a block can only be finalized after multiple subsequent blocks have been produced by different committees, implicitly vouching for the correctness of the result). Alternatively, systems

such as Steward [1], Fireplug [25], ResilientDB [17] or Multi-Layer PBFT [23] organize processes in hierarchical groups to achieve low message complexity and balance the bandwidth load. However, they sacrifice resilience by tolerating significantly less than $f = \frac{N-1}{3}$ failures.

Approaches such as Byzcoin [20], Motor [19], and Omniledger [21] address the bottleneck at the leader by organizing processes in a tree topology, with the leader placed at the root. The tree is used to disseminate information from the leader to the other processes, and to aggregate votes using cryptographic schemes such as multisignatures [4, 20]. The use of trees reduces the number of messages any single process has to send, receive, and process (that becomes logarithmic with the system size) by distributing the load among all internal nodes of the tree.

The use of trees comes, however, at the cost of an increased latency of each consensus round. In fact, while in PBFT a round can be executed within a single communication step, in HotStuff it requires two communication steps (i.e., a roundtrip), and in trees it requires $2h$ communication steps, where h is the height of the tree. If the blockchain protocol only starts a new consensus instance after the previous one terminates, this increase of the per-round latency has a direct negative impact on the system throughput. In fact, the advantages that stem from the load distribution can easily be outweighed by the disadvantages associated with longer consensus rounds. Strikingly, neither Motor nor Omniledger discuss or mitigate the impact of the additional latency on the throughput resulting from the increased number of communication steps required to complete each round.

Pipelining allows to mitigate the negative impact on throughput of additional communication steps. In HotStuff the first round of the n^{th} consensus instance is executed in parallel with the second round of the $(n-1)^{th}$ consensus instance, and with the third round of the $(n-2)^{th}$ consensus instance, etc. This allows to piggyback information from multiple consensus instances in a single message. Unfortunately, pipelining increases the burden on the leader further amplifying the scalability limitations of a centralized approach.

Another disadvantage of trees is their slow recovery in the presence of faults. In approaches that use a clique or a star topology, such as PBFT or HotStuff, respectively, the system is able to make progress as long as the leader is non-faulty. Moreover, if the leader is faulty, the system is guaranteed to recover after $f + 1$ reconfigurations (also known as view changes in the literature). When using trees, progress is guaranteed if and only if all internal nodes of the tree are non-faulty (this is a sufficient but not necessary condition as discussed in §3). Furthermore, finding a configuration without faulty internal nodes has combinatorial complexity [19]. Due to these challenges, Byzcoin [20] quickly falls back to a clique when faults occur. Motor [19] and Omniledger [21] build upon the principles of Byzcoin, but rather than falling back immediately to a clique topology, rotate the nodes in the subtrees in an attempt to let the leader, i.e. the root of the tree, gather $N - f$ signatures. If, in a given round, the root process is unable to collect a quorum of signatures, it contacts directly a random subset of leaf processes, which in turn will attempt to collect votes from their siblings, until a quorum is obtained. Thus, in the worst case scenario, assuming a fanout of m, after $\frac{N}{m}$ steps the root will contact every other node directly, as if the system was using a star topology. If the root itself fails during this process, a new tree is formed but, if more faults occur, the entire procedure may need to be repeated. Furthermore, this strategy only works with trees with a maximum height of $h = 2$.

Table 1. Comparison of existing Algorithms

	Load Balancing for Scalability	Deterministic finality/ $N = 3f + 1$ resilience	Throughput independent of round latency	Quick recovery
PBFT [7]	✗	✓	✗	✓
HotStuff [33]	✗	✓	✗	✓
Steward [1]	✓	✗	✗	✗
Fireplug [25]	✓	✗	✗	✗
ResilientDB [17]	✓	✗	✓	✗
Multi-Layer [23]	✓	✗	✗	✓
Byzcoin [20]	✓	✓	✗	✗
Omniledger [21]	✓	✓	✗	✗
Algorand [15]	✓	✗	✗	✗
SCP [24]	✓	✗	✗	✗
Kauri (this paper)	✓	✓	✓	✓

Table 1 summarizes our discussion of the systems based on the criteria discussed above. The table highlights that no previous system leverages load balancing techniques to promote scalability while preserving high resilience and high throughput. Furthermore, most systems that achieve some form of load balancing either decrease fault tolerance or increase the complexity of the reconfiguration leading to a slow recovery under faults.

In this paper we propose Kauri, a BFT communication abstraction that leverages dissemination/aggregation trees for load balancing and scalability while avoiding the main limitations of previous tree-based solutions, namely, poor throughput due to additional round latency and the collapse of the tree to a star even in runs with few faults.

Kauri introduces novel pipelining techniques suitable for trees of arbitrary depth that sustain high throughput as the system grows in size. As in HotStuff, Kauri starts a new instance of consensus before the previous instance has terminated. But, unlike HotStuff, Kauri starts a new round while the previous round is still being propagated in the tree, effectively exploiting the potential parallelism created by the different stages (one stage per height) of the tree. This allows the leader to effectively use the available bandwidth without becoming a bottleneck. One of the key challenges behind our combination of trees and pipelining is that using arbitrary pipeline values results in poor performance: under-pipelining fails to take advantage of the available parallelization opportunities, while over-pipelining congests the system - hence, simply using HotSuff's star-based pipelining

in Kauri trees would not yield good results. We overcome this challenge by introducing a performance model that approximates, for a given scenario, the ideal pipelining values that maximize performance.

We also introduce a novel reconfiguration strategy that, when the number of consecutive faults is small (arguably the most common case), continues to use a tree topology rather than falling back to a star. More precisely, for a tree fanout of m, if $f < m$, Kauri can find a robust tree-based configuration in f+1 reconfiguration steps (which is optimal), and fall back to a star topology only in runs where $f \geq m$ consecutive faults occur. Thus, Kauri offers the same fault-tolerance of traditional approaches (i.e. ensures correctness as long as $f < N/3$), ensures deterministic finality (unlike committee-based solutions), and distributes the load among internal nodes allowing it to scale with the number of nodes and achieve high throughput. We implemented Kauri and evaluated it under different realistic scenarios with up to 400 processes. Results show that Kauri outperforms HotStuff's throughput by up to 28x.

In short, the paper makes the following contributions: i) We present a set of abstractions that support the use of aggregation/dissemination trees in the context of consensus protocols; ii) We introduce a performance model that shows how pipelining can be used to fully leverage the parallelisation opportunities offered by the trees; iii) We present a precise sufficient condition that allows efficient reconfiguration of the tree without falling back immediately to a star topology; iv) We present Kauri, the first tree-based communication abstraction for BFT consensus protocols that achieves higher throughput than HotStuff in all considered scenarios and better latency under certain conditions; v) We present an extensive experimental evaluation of Kauri in realistic scenarios with up to 400 nodes.

2 System Model

We assume the system is composed of N server processes $\{p_1, p_2, \ldots, p_N\}$ and a set of client processes $\{c_1, c_2, \ldots, c_m\}$. We also assume the existence of a Public Key Infrastructure used by processes to distribute the keys required for authentication and message signing. Moreover, processes may not change their keys during the execution of the protocol and require a sufficiently lengthy approval process to re-enter the system to avoid rogue key attacks [27]. We assume the Byzantine fault model, where at most $f < N/3$ faulty processes may produce arbitrary values, delay or omit messages, and collude with each other, but do not possess sufficient resources to compromise the cryptographic primitives.

Processes communicate via perfect point-to-point channels with the following properties: *Validity:* If a process p_j delivers a value v on a channel over an edge e_{ij}, v was sent by p_i. *Termination:* If both p_i and p_j are correct, if p_i invokes SEND then eventually p_j delivers v. These are implemented using mechanisms for message re-transmission and detection and suppression of duplicates [6]. To circumvent the impossibility of consensus[14], we assume the partial synchrony model [10]. In this model, there may be an unstable period, where messages exchanged between correct processes are arbitrarily delayed. However, there is a known bound Δ on the worst-case network latency and an unknown Global Stabilization Time (GST), such that after GST, all messages between correct processes arrive within Δ. Note that safety is always preserved and the partial synchrony assumptions are necessary only to ensure liveness [33].

3 Using Dissemination/Aggregation Trees

Instead of designing a completely new consensus algorithm from scratch, we developed Kauri as an extension of HotStuff. The key idea is to replace the dissemination and aggregation patterns used by HotStuff, which are based on a star topology, by new patterns based on tree topologies. While for simplicity our presentation hinges on HotStuff characteristics, our principles could also be applied to other leader-based consensus algorithms.

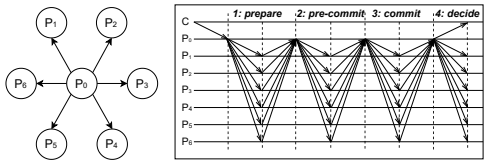
3.1 HotStuff Communication Pattern

For self-containment, we provide a brief high-level description of HotStuff. We give emphasis on the communication pattern used in HotStuff and discuss how this pattern may be abstracted, such that it can be replaced by different implementations. HotStuff reaches consensus in four communication rounds. Each round consists of two phases: i) a dissemination phase where the leader broadcasts some information to all processes; and ii) an aggregation phase where the leader collects and aggregates information from a quorum of replicas. All rounds follow the same exact pattern, but the information sent and received by the leader in each round differs:

First round: In the dissemination phase, the leader broadcasts a block proposal to all processes. In the aggregation phase, the leader collects a *prepare* quorum of $N - f$ signatures of the block. The signatures convey that the replicas have validated and accepted the block proposed by the leader.

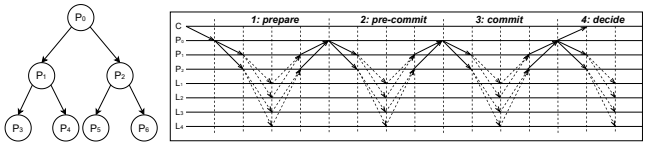
Second round: In the dissemination phase, the leader broadcasts the *prepare* quorum. In the aggregation phase, the leader collects the *pre-commit* quorum, including $N - f$ signatures from processes that have validated the *prepare* quorum. If the leader is able to collect a *pre-commit* quorum, the value proposed by the leader is *locked* and will not be changed, even if the leader is subsequently suspected.

Third round: In the dissemination phase, the leader broadcasts the *pre-commit* quorum. In the aggregation phase, the leader collects a *commit* quorum, including $N - f$ signatures of processes that have validated the *pre-commit* quorum. If the leader is able to collect the quorum, the value is decided.



(a) Topology **(b)** Communication pattern

Figure 1. HotStuff communication pattern with 7 processes.



(a) Topology **(b)** Communication pattern

Figure 2. Tree communication pattern for 7 processes.

Fourth round: In the last round, the leader broadcasts the *commit* quorum to all processes, which in turn verify it and decide accordingly.

Figure 1 illustrates the communication pattern of HotStuff in a system with 7 processes. HotStuff uses the following two primitives:

- *broadcastMsg(data).* This primitive is used in the first phase of each round to broadcast *data* from the leader to all other processes.
- *waitFor (N - f) votes.* This primitive is used in the second phase of each round, for the leader to collect votes from a quorum of $N - f$ processes.

The implementation of these primitives must satisfy the following properties:

Definition 1. ***Reliable Dissemination:*** *After the GST, and in a robust configuration, all correct processes deliver the data sent by the leader.*

Definition 2. ***Fulfillment:*** *After the GST, and in a robust configuration, the aggregate collected by the leader includes at least $N - f$ votes.*

It is easy to show that, when using perfect point-to-point channels, it is possible to achieve Reliable Dissemination and Fulfillment on a star topology. For this purpose, we first define the notion of a *robust star.*

Definition 3. ***Robust Star:*** *A star is said to be robust if the leader is correct, and non-robust if the leader is faulty.*

Briefly, the assumption of perfect point-to-point channels and the fact that the leader is correct ensure that all correct processes deliver the message sent by the leader, hence satisfying Reliable Dissemination. In a similar fashion, all correct processes are able to send their vote to the leader which, in turn, is able to collect $N - f$ votes/signatures and hence satisfy Fulfillment. For lack of space, we refer the reader to the HotStuff paper for full details [33].

3.2 Using Trees to Implement HotStuff

We now discuss how to implement the *broadcastMsg* and *waitFor* primitives using tree topologies, while preserving the same properties. As noted before, processes are organized in a tree with the leader at the root. The primitive *broadcastMsg* is implemented by having the root send *data* to its children that in turn forward it to their own children, and so forth. The primitive *waitFor* is implemented by having the leaf nodes send their signatures to their parent. The parent then aggregates those signatures with its own and sends the aggregate to their own parent. This process is repeated until the final aggregate is computed at the root of the tree. This process is illustrated in Figure 2.

When using a tree to implement *broadcastMsg* and *waitFor*, the notion of robust configuration needs to be adapted, as it is no longer enough that the leader is non-faulty to make the configuration robust. We define a *robust tree* as follows.

Definition 4. ***Robust Tree:*** *An edge is said to be safe if the corresponding vertices are both correct processes. A tree is robust iff the leader process is correct and, for every pair of correct process p_i and p_j, the path in the tree connecting these processes is composed exclusively of safe edges.*

Note that our definition of a robust tree is a sufficient but not necessary condition to achieve consensus. In fact, consensus can be reached as long as there is a path composed exclusively of safe edges between the leader and a quorum of correct processes. Our simpler formulation discards some viable configurations - for instance a tree with a faulty internal node where all its children are also faulty - but provides an important corollary: a tree is robust if and only if all internal nodes, including the leader, are correct processes. This observation allows us to devise an efficient reconfiguration algorithm that is optimal when the number of consecutive faults is small (§5).

3.3 Dissemination and Aggregation

We start by describing the communication primitives used to propagate information on the tree and the cryptographic primitives used to perform aggregation.

3.3.1 Communicating on the Tree. Processes use the tree to communicate. Each directed edge maps to a perfect single-use point-to-point channel used to send and deliver a single value. Note that, when using perfect channels, a message is only guaranteed to be eventually delivered if both the sender and the recipient are correct. If the sender is faulty, no message may ever be delivered. To avoid blocking, a process should be able to make progress if a message takes too long to be received. Moreover, the single-use ensures that the receiver either returns the current value sent or $\perp$

Algorithm 1 Impatient Channels: RECEIVE

```
1: let ic be an impatient channel built on top of perfect channel pc
2: function IC.RECEIVE( p)                                  ▷ receive from p
3:     TIMER.START(Δ)
4:     when pc.DELIVER (p, v) do return v
5:     when TIMER.TIMEOUT() do return ⊥
6: end function
```

and never older values. In practice, this can be achieved by assigning a unique identifier to each instance and tagging the corresponding messages with this identifier.

This behavior is captured by an abstraction we call *impatient channels*. Impatient channels offer a blocking RECEIVE primitive that always returns a value: either the value sent by the sender, or a special value $\perp$ if the sender is faulty or the system is unstable. After the GST, if the sender and the receiver are correct, the receiver always receives the value sent. Impatient channels have the following properties:

- *Validity:* If a process p_j delivers a value v on a channel over an edge e_{ij}, v was sent by p_i or $v = \perp$.
- *Termination:* If a correct process p_j invokes RECEIVE, it eventually returns some value.
- *Conditional Accuracy:* Let p_i and p_j be correct sender and receiver processes, respectively. After GST, p_j always return the value v sent by p_i.

Algorithm 1 shows how impatient channels can be implemented on top of perfect channels using the known bound Δ on the worst-case network latency.

3.3.2 Cryptographic Collections. In each round of consensus, it is necessary to collect a Byzantine quorum of votes. The collection and validation of these votes can be an impairment for scalability. Kauri mitigates these costs by using the tree to aggregate votes as they are forwarded to the leader. We model the process of vote aggregation with a *cryptographic collection* abstraction that corresponds to a secure multi-set of tuples (p_i, v_i). A process p_i can create a new collection c with a value v_i by calling c=NEW$((p_i, v_i))$. Processes can also merge two collections using a combine primitive denoted by $c_{12} = c_1 \oplus c_2$. A process can also check if a collection c includes at least a given threshold of t distinct tuples with the same value v, by calling HAS(c, v, t). Finally, it is possible to check the total number of input tuples combined in c by checking its cardinality $|c|$. Cryptographic collections have the following properties:

- *Commutativity:* $c_1 \oplus c_2 = c_2 \oplus c_1$
- *Associativity:* $c_1 \oplus (c_2 \oplus c_3) = (c_1 \oplus c_2) \oplus c_3$
- *Idempotency:* $c_1 \oplus c_1 = c_1$
- *Integrity:* Let $c = c_1 \oplus \ldots c_i \ldots c_n$. If HAS (c, v, t) then at least t distinct processes p_i have executed c_i=NEW$((p_i, v))$

Note that different cryptographic techniques can be used to implement these collections. In Kauri, we leverage a non-interactive BLS multisignature scheme that allows each internal node to aggregate the votes from its children into one single aggregated vote [4]. The burden imposed on each internal node (including the root) is $O(m)$, where m is the fanout of the tree and the complexity of verifying an aggregated vote is $O(1)$. Note that classical asymmetric signatures require $O(N)$ verifications at each process [4].

Algorithm 2 *broadcastMsg* on a tree T (process p_i)

```
1: procedure BROADCASTMSG(T, data)
2:     children ← T.CHILDREN(p_i)          ▷ Get edges to children of p_i
3:     parent ← T.PARENT(p_i)              ▷ Get parent of p_i (returns ⊥ for root)
4:     if parent ≠ ⊥ then
5:         data ← IC.RECEIVE(parent)       ▷ Receive from parent
6:     end if
7:     for all e ∈ children do             ▷ Send to children
8:         IC.SEND(e, data)
9:     end for
10:    return data
11: end procedure
```

3.3.3 Implementing *broadcastMsg*. The implementation of *broadcastMsg* on a tree is presented in Algorithm 2. Note that the algorithm always terminates, even if some intermediate nodes are faulty. This is guaranteed since impatient channels always return a value after the known bound Δ on the worst-case network latency, either the *data* sent by the parent or the special value $\perp$.

Theorem 1. *Algorithm 2 guarantees Reliable Dissemination.*

Proof. We prove this by contradiction. Assume Reliable Dissemination is not guaranteed. This implies that at least one correct process did not receive the data sent by the leader. This is only possible if: i) at least one correct process is not connected to the leader either directly or through correct intermediary processes, ii) one of the intermediary processes or the root process did not invoke CHANNEL.SEND for at least one correct child process, or iii) the data got lost in the channel. Reliable Dissemination is defined only for a robust configuration which, following the definition of a Robust Tree, ensures that the leader is correct and there is a path of correct processes between the leader and any other correct process. Thus, the first case is not possible. Moreover, correct processes follow the algorithm and, because correct processes can only have correct parents in a robust configuration, the second case is also impossible. Finally, the third case is also impossible due to the use of perfect channels. Therefore, Algorithm 2 guarantees Reliable Dissemination. □

3.3.4 Implementing *waitFor*. Algorithm 3 presents the implementation of *waitFor* on a tree. The algorithm relies on the cryptographic primitives to aggregate the signatures as they are propagated toward the root. Like *broadcastMsg*, *waitFor* always terminates, even if some nodes are faulty. This is guaranteed because impatient channels always return a value after the known bound Δ on the worst-case network latency, either the *data* sent by the child processes or the special value $\perp$. Before GST or in non-robust configurations, the collection returned at the leader may be empty or include just a subset of the required signatures.

Theorem 2. *Algorithm 3 guarantees Fulfillment.*

Algorithm 3 *waitFor* on a tree T (process p_i)

```
1:  procedure WAITFOR(T, input)
2:      children ← T.CHILDREN(p_i)              ▷ Get edges to children of p_i
3:      parent ← T.PARENT(p_i)                  ▷ Get parent of p_i (returns ⊥ for root)
4:      collection ← NEW((p_i, input))
5:      for all e ∈ children do                 ▷ Empty for leaf nodes
6:          partial ← IC.RECEIVE(e)
7:          collection ← collection ⊕ partial
8:      end for
9:      if parent ≠ ⊥ then
10:         IC.SEND(parent, collection)
11:     end if
12:     return collection
13: end procedure
```

Proof. We prove this by contradiction. Assume that the leader process was unable to collect $N-f$ signatures. Following Algorithm 3, this means that either: i) an internal node did not receive the signatures from all correct children (line 6), ii) or an internal node did not aggregate and relay the signatures it has received from its correct children (line 10). Since we assume impatient channels, that are implemented on top of perfect point-to-point channels, the first case is not possible after GST. The second case may happen, if either the internal node omits signatures in the aggregate, does not relay any signatures, or is blocked waiting indefinitely for messages from its children. Either option leads to a contradiction. Since we assume a robust graph, all internal nodes between the root and a correct process must be correct and hence follow the algorithm. Additionally, due to the impatient channels, eventually each channel will return a value to the internal node making sure that eventually it will unblock and relay all collected signatures from all correct child processes. Therefore Algorithm 3 guarantees Fulfillment. □

3.3.5 Challenges of Using a Tree.

The implementations of the *broadcastMsg* and *waitFor* primitives that we introduced above allow us to replace the star topology used in HotStuff with a tree topology that is more efficient and scalable as we show in the evaluation (§7). However, two remaining challenges need to be addressed to make the tree topology a valid alternative in practice:

Mitigate the increased latency: While trees allow to distribute the load among all processes, the additional round-trip of the *broadcastMsg* and *waitFor* primitives result in additional latency, which might negatively affect system throughput. We discuss how to mitigate this in §4.

Reconfiguration strategy: In HotStuff, the configuration is robust if the leader is non-faulty. Therefore, there are only f non-robust configurations and $N-f$ robust configurations. It is thus trivial to devise a reconfiguration strategy that yields a robust configuration in an optimal number of steps (i.e. $f+1$). In a tree, a configuration is robust iff the root and all internal nodes are correct. The total number of configurations and the subset of non-robust configurations is extremely large. In §5 we introduce a reconfiguration strategy that builds robust configurations in a small number of steps.

4 Mitigating Tree Latency

In this section we introduce the mechanisms to mitigate the additional latency inherent to tree topologies when compared to HotStuff's star topology. As described earlier, HotStuff needs four communication rounds for each instance of consensus. If HotStuff waited for each consensus instance to terminate before starting the next one, the system throughput would suffer significantly. Therefore, HotStuff relies on a pipelining optimization, where the $i+1$ instance of consensus is started optimistically, before instance i is terminated. As a result, at any given time, each process participates in multiple consensus instances. Furthermore, to reduce the number of messages, HotStuff combines the information of these parallel consensus instances in a single message.

By following the same structure, Kauri is amenable to the same optimization. However, because Kauri uses a tree, the latency to terminate a given round (and hence a consensus instance) is substantially larger than in HotStuff. While at first this might look like an obstacle, it opens the door for more advanced pipelining techniques that substantially improve throughput and hide the additional latency induced by trees. In fact, as we will show in the evaluation (§7), in certain scenarios, Kauri can achieve not only higher throughput, but also lower latency than HotStuff.

4.1 Pipelining in HotStuff

We start by providing an overview of HotStuff's pipelining using the seven node scenario previously introduced in Figure 1. Figure 3 illustrates the execution of multiple rounds of consensus in HotStuff where each round is depicted in a different shade of gray.

Consider the first round (light gray) that starts with the leader sending the block to all other processes (downward arrows). The time this step takes depends on the size of the data being transmitted, the available bandwidth, and the total number of nodes. To conclude a round, the leader has to collect a quorum of signatures. These signatures start flowing toward the leader as soon as the first process receives the message from the leader (upward arrows). Therefore, in a given round, dissemination and aggregation are partially executed in parallel. Soon after the dissemination of a round finishes, the leader may already start the next round of consensus.

To implement pipelining, HotStuff optimistically starts a new instance of consensus by piggybacking the first round messages of the next consensus instance with the second round messages of the previous instance. Because HotStuff requires four rounds of communication, this process can be repeated multiple times, resulting in messages that carry information of up to four pipelined consensus instances. In HotStuff the pipelining depth (i.e. the maximum number of consensus instances that can run in parallel) is thus equal to the number of communication rounds.

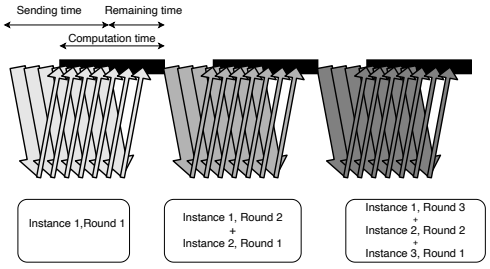


Figure 3. Pipelining in HotStuff

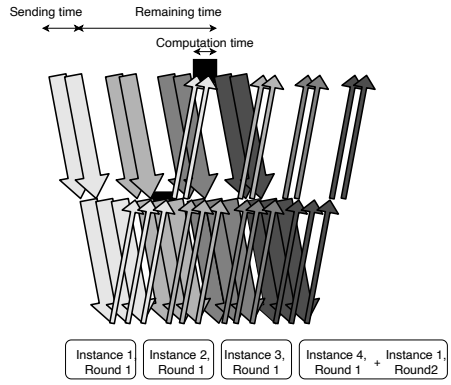


Figure 4. Pipelining in Kauri

4.2 Pipelining in Kauri

In Kauri, we extend pipelining to fully leverage the load balancing properties of trees as illustrated in Figure 4. In a tree, the fanout m is much smaller than the number of nodes N, and therefore in Kauri the root completes its dissemination phase much faster than in HotStuff, and it may become idle long before it starts collecting votes. This allows the root to start multiple instances of consensus during the execution of a single consensus round. This introduces a multiplicative factor that we call the pipelining *stretch* that augments the pipelining depth of HotStuff. In the example of Figure 4, the leader is able to start 3 new instances during the execution of the first round of a given consensus instance. Note that, in this example, the messages from the second round of instance 1 are piggybacked with messages from the first round of instance 4, i.e. a message carries information from consensus instances/rounds that are farther away in the pipeline. This increase in the pipelining depth allows for a higher degree of parallelism, and hence throughput.

4.3 Pipelining Stretch and Expected Speedup

Kauri's pipelining stretch, i.e. the number of instances that can be initiated during a single round, is affected by the following parameters:

Sending time: the time a node takes to send a block to all its children. This value is a function of the fanout m, the block size B, and the link bandwidth b, and is approximated by $\frac{mB}{b}$.

Processing time: the time a node takes to validate and/or aggregate the votes it receives from its children. This heavily depends on the type of signatures used by the algorithm. We measured these values experimentally, for different signature schemes (see §7).

Remaining time: the time that elapses from point the root finishes sending the block to its children until it receives and processes the last reply. This value is a sum of the network latency and the processing time as defined above. It is roughly given by:

$$remaining\ time = h \cdot (RTT + processing\ time)$$

where h is the height of the tree and RTT is the network roundtrip time. In a star topology the remaining time is small and mainly used to collect and process replies. However, in a tree, the root is often idle for most of the remaining time.

Kauri leverages this larger *remaining time* to start additional consensus instances. The challenge is therefore to estimate how many additional instances can be started, i.e. to estimate the pipelining stretch. For presentation simplicity, we assume that sending and processing can be performed concurrently. In a system where the bottleneck is the bandwidth, i.e. where the *sending time* is much larger than the *processing time*, the number of additional consensus instance that can be started during the *remaining time* is given by: $\frac{remaining\ time}{sending\ time}$. Similarly, in a system where the bottleneck is the CPU, i.e. where the *processing time* is much larger than the *sending time*, the number of additional consensus instances that can be started during the *remaining time* is given by: $\frac{remaining\ time}{processing\ time}$.

Kauri's pipelining stretch allows us to make the best use of the time the leader saves by having to interact with just m nodes instead of $N-1$ nodes. Therefore, the ratio between $N-1$ and m defines the maximum speedup we can achieve by using a tree instead of a star. For instance, in a system of 400 nodes, organized in a tree with fanout 20, the maximum speedup we can expect Kauri to offer is 19.95.

5 Reconfiguration

We now discuss Kauri's reconfiguration strategy. Recall that, in Kauri, processes use a tree to communicate. Due to faults or an asynchronous period, the tree may be deemed not robust and therefore a reconfiguration procedure is necessary to build a new tree. Naturally, not all possible trees are robust and several reconfigurations might be necessary before a robust tree is found.

Note that any leader-based protocol may require $f+1$ reconfigurations to find a robust topology, given that f consecutive leaders may be faulty. Our challenge is to avoid making the reconfiguration of Kauri superlinear with the number of processes, while also avoiding to fall back immediately to a star topology as soon as a single fault occurs.

We conjecture that building a general reconfiguration strategy that yields a robust tree in a linear number of steps without falling back to a star topology is impossible, due to the large number of non-robust configurations that may occur in a tree. Consider, for instance, the case of binary trees where the number of possible binary trees is given by the Catalan number $C_N = \frac{(2N)!}{(N+1)!N!}$. From all these trees, only a small fraction is robust, namely those where faulty processes are not internal nodes (Definition 4). Thus, a reconfiguration strategy that considers all possible configurations may require a factorial number of steps to find a robust tree. We discuss our reconfiguration strategy next.

5.1 Modeling Reconfiguration as an Evolving Graph

We model the sequence of trees as an evolving graph, i.e. a sequence of static graphs (that are trees). To ensure that eventually a robust tree is used, the evolving graph must observe the following property:

Definition 5. ***Recurringly Robust Evolving Graph:*** *An evolving graph $\mathcal{G}$ is said to be recurringly robust iff robust static graphs appear infinitely often in its sequence.*

A recurringly robust evolving graph is sufficient to ensure that a robust graph will eventually be used by processes to communicate. However, this is undesirable in practice because the number of reconfigurations until a robust graph is found is unbounded. Because the system is essentially halted during reconfiguration, we would like to find a robust graph after a small number t of reconfigurations. We call this property of an evolving graph *t-Bounded conformity.*

Definition 6. ***t-Bounded Conformity****: a recurringly robust evolving graph $\mathcal{G}$ exhibits t-Bounded conformity if a robust static graph appears in $\mathcal{G}$ at least once every t consecutive static graphs.*

5.2 Achieving t-Bounded Conformity

We now introduce an algorithm that achieves t-Bounded Conformity. We split all processes into t disjoint bins, each of size greater or equal to I, where I is the number of internal nodes in a tree. Then we build an evolving graph by creating trees whose internal nodes are drawn exclusively from a given bin, i.e., each tree T^k is built by picking a bin B^i, following a round robin strategy, and by assigning nodes from bin B^i to all internal nodes of the tree (including the root) as depicted in Algorithm 4.

Theorem 3. *Algorithm 4 constructs an evolving graph that satisfies t-Bounded Conformity as long as $f < t$.*

Proof. Because the t bins are disjoint and there are at most $f < t$ faults, at least one of the bins is composed exclusively of correct nodes. In each t consecutive steps, the algorithm picks a tree whose internal nodes are drawn from distinct bins, hence guaranteeing that a robust tree is found. □

Algorithm 4 Construction of an Evolving Tree with t-Bounded Conformity

1: Initially, split the set of processes $\mathcal{N}$ into $t+1$ disjoint bins
2: $\mathcal{N} \leftarrow B^0 \cup B^1 \cup \ldots \cup B^{\frac{N}{T}}$; s.t. $|B^i| \geq f+1 \wedge B^i \cap B^j = \emptyset$.
3: **function** BUILD(k)
4: $\quad i \leftarrow k$ **mod** $(t+1)$
5: $\quad \mathcal{G}^i \leftarrow$ all possible trees whose internal nodes are drawn exclusively from B^i.
6: $\quad T^k \leftarrow$ pick any tree from $\mathcal{G}^i$
7: $\quad$ **return** T^k
8: **end function**

A limitation of the approach is that each bin must be large enough to contain at least as many processes as the number of internal nodes of a tree. This limits the number of bins that can be generated and, therefore, the maximum value for t. In a balanced tree of fanout m we can, at most, obtain m disjoint bins with enough capacity to fill all the internal positions, such that this algorithm allows us to achieve at most $(m-1)$-Bounded Conformity.

5.3 Gracefully Degraded Reconfiguration

Given that Algorithm 4 can only achieve $(m-1)$-Bounded Conformity, if $f \geq m$ we will not be able to reconfigure the system in an optimal number of steps. Therefore, we adopt the following pragmatic approach to reconfiguration: we execute Algorithm 4 and if after m steps a robust tree is not found, Kauri falls back to a star. Thus, in the worst case scenario, Kauri performs $m + f + 1$ reconfigurations until a star with a non-faulty leader is found.

When the number of actual faults is small, i.e. if $f < m$, Kauri recovers quickly without losing its scalability and throughput properties. If the number of faults is large, i.e. if $m \leq f < N/3$, the system still recovers in a linear number of steps, but falls back to the performance of the original (star based) HotStuff protocol.

6 Implementation

We have implemented Kauri by extending the publicly available HotStuff implementation[1]. The core of the effort was extending the code to include the implementation of the *broadcastMsg* and *waitFor* primitives, as specified in Algorithm 2 and Algorithm 3, respectively. We also added support for the BLS cryptographic scheme by including the publicly available implementation used in the Chia Blockchain [4, 8]. This allows internal nodes to aggregate and verify the signatures of their children, and thus balance the computational load. Both the verification cost of the signature aggregates, and the size of aggregates have a small $O(1)$ complexity, contributing to the overall efficiency of the implementation.

Because HotStuff and Kauri share the same codebase, we also implemented a variant of HotStuff that uses the BLS signature scheme. As we discuss in more detail in the next section, this allows us to assess the effects of our contributions versus simply adopting another cryptographic scheme

[1] Available at https://github.com/hot-stuff/libhotstuff

in HotStuff. We denote the original HotStuff implementation as *HotStuff-secp* and the BLS variant as *HotStuff-bls*.

Pipelining: To implement pipelining, we use an estimation of the parameters discussed in §4 to compute the ideal time to start the dissemination phase for the next consensus instance. In the current implementation, we use a static pre-configured value but this could be automatically adapted at runtime, which we leave for future work.

Reconfiguration: To trigger reconfigurations we leverage the existing HotStuff mechanisms. In detail, if no consensus is reached after a timeout, each process compiles a *new-view* message that includes the last successful quorum it observed and sends it to the next leader [33]. In turn, the candidate leader waits for $2f+1$ *new-view* messages and, depending on the collected information, either continues the work of the previous leader (if a block was previously locked) or proposes its own block (if no block had been locked yet). Similarly, in Kauri and upon timeout, each process invokes the *build* function of Algorithm 4 to construct the next tree and sends the same *new-view* message to the root of the new tree. The root then also awaits $2f + 1$ *new-view* messages before re-initiating the protocol. Note that the *new-view* messages do not use the tree and are instead sent directly to the candidate leader as in HotStuff. This is the only time in Kauri where all processes communicate directly with the leader.

Code Availability: Overall, the implementation of the functionalities described above required the addition/adaptation of $\approx$ 1300 loc to the HotStuff codebase[2].

7 Evaluation

In this section we evaluate Kauri across several scenarios.

7.1 Experimental Setup

All experiments were performed on the Grid'5000 testbed [12]. We used 20 physical machines, each with two Intel Xeon E5-2620 v4 8-core CPUs and 64 GB RAM. We deploy a variable number of processes (from 100 to 400) in these machines. As we will discuss later, in some configurations Kauri is able to saturate the hardware resources of our testbed.

We evaluate Kauri on a wide range of deployments, that capture the different scenarios where we believe that permissioned blockchains with a large number of participants are likely to be used. More precisely, we consider the following deployment scenarios: *global*, *regional*, and *national*. The *global* deployment models a globally distributed blockchain as used in other works [15, 19, 20] with 200ms roundtrip time (RTT) and 25Mb/s bandwidth. The two other scenarios model reported industry use cases [9] in more limited geographical deployments such as local supply-chain management. The *regional* deployment captures a deployment in a large country or unions of countries, such as the US or the EU, with 100ms RTT and 100Mb/s bandwidth. The *national* deployment models a setting where nodes are closer to each other with 10ms RTT and 1000Mb/s bandwidth. Finally, we also consider a *heterogeneous* deployment with a mix of different bandwidth and RTT characteristics, as used in other recent works[17]. We model the network characteristics of each scenario using NetEm[18].

For most experiments, we use three system sizes with $N = 100, 200, 400$ processes. As expected in most realistic deployments, these values of N do not yield perfect m-ary trees. Therefore, we simply assign processes to tree nodes such that it approximates a balanced tree. Unless otherwise stated, this results in the following trees: for $N = 100$, $m = 10$ for the root and $m = [8, 9]$ for the other internal nodes; for $N = 200$ the root's fanout is $m = 14$, and for the other internal nodes $m = [13, 14]$, and for $N = 400$, $m = 20$ for the root and $m = [18, 19]$ for the remaining internal nodes. This results in trees with height 2 that are used throughout the experiments, unless otherwise stated.

7.2 Configuring Kauri

We now describe the values that are used to configure Kauri. Table 2 shows the values of the different parameters required to compute the Kauri pipelining stretch, following the rationale introduced in §4.3. We consider different bandwidth/RTT scenarios and blocks of 250Kb (plus signatures). For each configuration we present the *processing time*, which we have measured experimentally, the *sending time*, and the *remaining time* (§4.3). These values are used to compute both the target pipelining stretch and the expected maximum speedup for each configuration. Note that, due to space constraints, Table 2 does not include the parameter values for all configurations we evaluate in the following sections. For instance, we have also experimented with block sizes different than 250Kb. The purpose of the table is not to be exhaustive but to offer a conceptual framework that makes it easier to reason about the experimental results presented in the next sections.

7.3 Effect of Pipelining Stretch on Throughput

We start by showing the effects of the pipelining stretch on Kauri's throughput.

Figure 5 depicts the throughput achieved when using Kauri in a setting with $N = 100$ in the *global* scenario (200ms RTT and 25Mb/s bandwidth) for different block sizes and increasing pipelining stretch values.

For a blocksize of 250Kb, the experimental numbers are close to the numbers predicted by our model and presented in Table 2 (3rd line for Kauri), i.e. the best results are achieved with a pipelining stretch close to 5. This shows that our performance model, albeit simple, can offer a good estimate of the performance of the real system. The figure also shows that, with smaller block sizes, higher pipelining stretch values are needed to make full use of the resources. This is also

[2]Available at https://github.com/Raycoms/Kauri-Public

Scenario	N	Height h	Root's fanout m	Processing time	Sending time	Remaining time	Base pipelining	Pipelining stretch	Total depth	Expected speedup
HotStuff-secp										
National	400	1	399	15	157	25	4	1	4	-
Regional	400	1	399	15	1569	115	4	1	4	-
Global	100	1	99	4	1153	204	4	1	4	-
Global	200	1	199	7	2591	207	4	1	4	-
Global	400	1	399	15	6277	215	4	1	4	-
Kauri										
National	400	2	20	27	5	54	4	3	12	≈ 6x
Regional	400	2	20	27	52	234	4	5	20	≈ 30x
Global	100	2	10	17	103	417	4	5	20	≈ 10x
Global	200	2	14	23	144	423	4	4	16	≈ 20x
Global	400	2	20	27	206	434	4	3	12	≈ 30x

Table 2. Pipelining stretch and estimated speedup vs HotStuff-secp for a block size of $250Kb$ and different N.

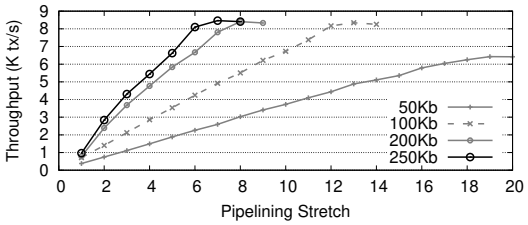


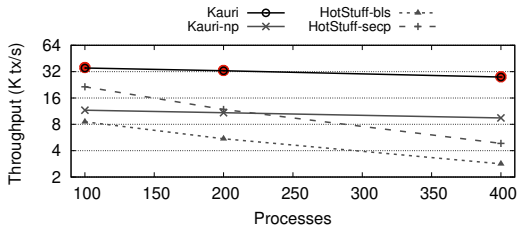
Figure 5. Effect of pipelining stretch on Kauri's throughput for $N = 100$, $m = 10$ and different block sizes.

expected given our model: with smaller block sizes the sending time is smaller and the idle portion of the remaining time is much larger. The ratio between these two values is also larger, thus allowing Kauri to start more instances while it waits for the responses from a previous instance. Naturally, it is more efficient to run fewer instances with more transactions each than many instances with a small number of transactions. For the rest of the evaluation, we use a blocksize of $250Kb$ for Kauri. For HotStuff, we empirically observed that a blocksize of $250Kb$ also yielded the best results across the different experiments.

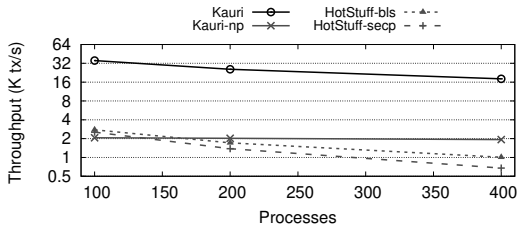
7.4 Throughput Across Different Scenarios

We now compare the throughput of Kauri with HotStuff-secp, the standard implementation, and our variant HotStuff-bls, in the national, regional and global scenarios. To assess the impact of our pipelining scheme we also include results for Kauri without pipelining (denoted as Kauri-np). By using trees with BLS signatures but without pipelining, Kauri-np captures the performance characteristics of existing non-pipelining tree based systems such as Motor [19] and Omniledger [21]. The results are depicted in Figure 6.

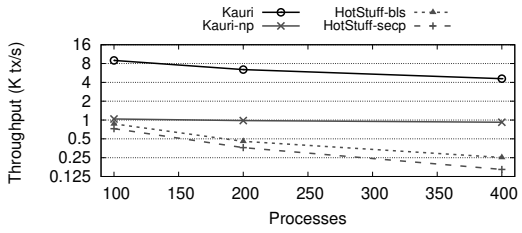
The first observation is that, as expected, HotStuff is extremely sensitive to the available bandwidth and to the total number of nodes in the system. The larger the number of nodes, the longer it takes for the leader to finish a given



(a) national ($10ms$ RTT - $1Gb/s$ links)



(b) regional ($100ms$ RTT - $100Mb/s$ links)



(c) global ($200ms$ RTT - $25Mb/s$ links)

Figure 6. Throughput for different configurations.

round. Also, for a fixed value of N, the *sending time* increases sharply as the network bandwidth decreases. As a result, the performance of HotStuff is highly penalized in systems with large numbers of participants and limited bandwidth. Also, since the use of BLS signatures reduces bandwidth usage, HotStuff-bls performs better than HotStuff-*secp* in all scenarios, except when using the $1Gb/s$ network.

A second observation is that, without our pipelining techniques, the performance of tree-based algorithms is mainly limited by the RTT. This is illustrated by Kauri-np where the throughput drops significantly when the RTT increases but only drops slightly with the number of processes. It is also interesting to observe that, as the network bandwidth decreases, even without our pipelining, the use of a tree already pays off. In fact, in the regional scenario, Kauri-np offers better throughput than HotStuff for a system with 200 or more processes.

Kauri, by leveraging the full capacity of the system via our pipelining mechanism, outperforms both HotStuff variants and Kauri-np in all scenarios. Interestingly, despite the simplifications adopted in our model (which, for instance, considers that computation and dissemination do not interfere with each other) the predicted speedup over HotStuff-secp is very close to the observed value. For instance, from Table 2

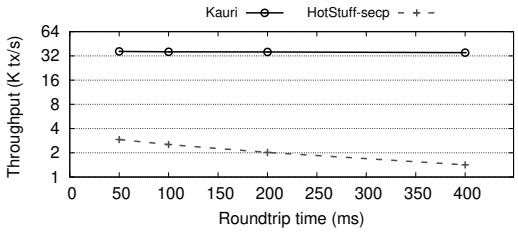


Figure 7. Impact of RTT in system throughput. (N=100 with $100Mb/s$ bandwidth)

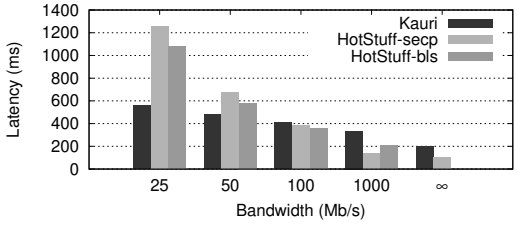


Figure 8. Bandwidth vs latency (N=100, RTT=100ms).

we expected a speedup over HotStuff-secp of approximately $\approx 30x$ for the system of 400 nodes in the global scenario and the value obtained experimentally is $28.2x$. This difference is quite reasonable given the number of simplifications adopted in the model. Overall, while the use of trees (Kauri-np) results in better performance than stars in certain scenarios, only the combination of trees and pipelining permits to achieve a substantial performance increase.

Note that the results of Kauri for the national scenario do not match the predictions of our model, which predicts a speedup of $\approx 6x$ with 400 processes. This is due to the limitations of our testbed that does not have capacity to support this experiment without incurring in a CPU bottleneck (note that due to the limited number of physical servers we are forced to collocate several process per server). In the plot and the rest of the evaluation we highlight the data points obtained in a saturated testbed with a red circle.

Finally, note that due to the use of a tree with a fixed height of $h = 2$, Kauri's throughput drops with the number of processes. In §7.8 we show how throughput can be preserved by increasing the height of the tree (and the pipelining stretch).

7.5 Effect of the RTT in Throughput

Pipelining not only allows Kauri to better exploit the available computing and network resources, but also contributes to mitigating the negative effects of additional RTT inherent in tree-based approaches. To further assess this, we conducted an experiment where we observe the throughput evolution as the RTT increases.

Figure 7 shows the results for N=100 in the regional scenario ($100Mb/s$ bandwidth) but where we varied the RTT from $50ms$ to $400ms$. As it can be observed, while the throughput of HotStuff-secp decreases as the RTT increases, the throughput of Kauri can be kept almost constant by increasing the pipelining stretch to avoid the leader from being idle while waiting for the replies. Following our model, the pipelining stretch varied from 7, for an RTT of $50ms$, to 33 for an RTT of $400ms$. Results for the other scenarios (not shown due to space constraints) follow a similar trend.

7.6 Latency

The previous experiments showed that Kauri, despite using longer rounds, can achieve better throughput than HotStuff due to the use of the pipelining stretch. We now conduct a similar study on latency.

Given that the use of a tree increases the latency required to exchange messages among the leader and the remaining processes, one could expect that Kauri would exhibit higher system latency than HotStuff. In fact, in a system with unlimited bandwidth and processing power, the latency of consensus is bound by the RTT. However, in realistic settings, bandwidth is not infinite and in practice, the system latency is limited by the *sending time*. The dissemination/aggregation parallelism enabled by a tree substantially reduces the *sending time*. This is particularly important in bandwidth constrained scenarios where the *sending time* has a much larger impact on latency than the additional number of communication hops required by a tree.

To confirm this we set up a scenario with a fixed RTT of $100ms$ and vary the bandwidth from $25Mb/s$ to $1000Mb/s$. Figure 8 shows the results for $N = 100$. As shown, the available bandwidth has a much larger impact on HotStuff-secp than in Kauri. In fact, for bandwidths smaller than $100Mb/s$ Kauri offers better latency than Hotstuff-secp, and only at high bandwidths HotStuff-secp starts to have substantially better latency. The pictures also shows (analytical) values for an idealized scenario of infinite bandwidth, where HotStuff-secp's latency would be at best half of Kauri's. As in §7.5, results for the other system sizes (not shown due to space constraints) follow a similar trend.

7.7 Throughput vs Latency

We now study the impact of load in the performance of the system. To this end we fix the system size, network bandwidth and latency, and vary the load in the system by manipulating the block size, i.e. the number of transactions offered by the client. Results are depicted in Figure 9 for the *global* scenario with $N = 100$ and the following block sizes: $32Kb$ $64Kb$, $125Kb$, $250Kb$, $500Kb$, and $1Mb$. For Kauri we adjust the pipelining stretch for each scenario following our performance model.

Following the observations of the previous experiments, the throughput of Kauri is substantially higher than that of both variants of HotStuff in all scenarios. As the block size increases, the latency of all systems increases as expected due to an increase in the *sending time*. This increase is however much faster in both variants of HotStuff, and for block

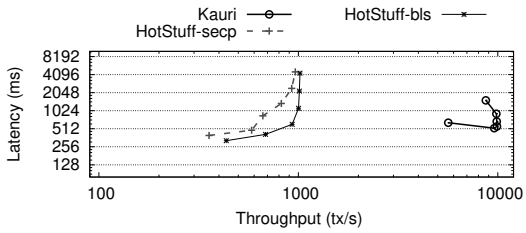


Figure 9. Througput-Latency for increasing block sizes (N=100, RTT=100ms, block size from 32Kb to 1Mb).

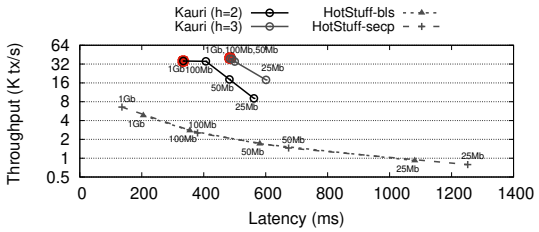


Figure 10. Throughput/Latency tradeoffs for different network bandwidth using trees of different heights

sizes larger than 125Kb HotStuff's latency surpasses that of Kauri. This highlights the importance of using a tree to spread the load and hence avoid a bottleneck at the leader process. HotStuff-bls outperforms HotStuff-secp in both latency and throughput for all scenarios, which also confirms the previous experiments when varying the available bandwidth (Figure 8). Finally, the decrease in latency in Kauri when going from a block sizes of 32Kb to 64Kb (the two first data points in the Kauri line) is due to pipelining effects on CPU usage. Blocks of 32Kb allows for a higher level of pipelining, which saturates the CPU. As the block size increases, the pipeline decreases, following our performance model (hence, gradually shifting the bottleneck away from the CPU and to the network).

7.8 Impact of Tree Height in Throughput/ Latency

The experiments of the previous sections have shown that the *sending time* is a key factor for both throughput and latency. By reducing the *sending time*, Kauri provides better throughput in all considered scenarios and better latency in bandwidth constrained scenarios.

We now study the impact of tree height on throughput and latency by setting an experiment with $N = 100$, an RTT of 100ms, and variable bandwidth. We deployed HotSuff-secp, HotStuff-bls (with $h = 1$ and $m = 99$), and two Kauri configurations: one with $h = 2$ and $m = 10$, as in the previous experiments, and another with $h = 3$ and $m = 5$.

Results are shown in Figure 10. The first observation is that by increasing the tree height it is possible to substantially increase Kauri's throughput, which almost doubles, with only a modest impact on latency. The results for Kauri with $h = 2$ at $1Gb/s$ bandwidth and $h = 3$ at $100Mb$ and above saturate our testbed, which is denoted by the red circle. According to our model, for $h = 2$ we expect the throughput to double when moving from a $100Mb/s$ bandwidth to $1Gb/s$. Similarly, for $h = 3$, we expect the throughput obtained with a bandwidth of $50Mb/s$ to double and quadruple with a bandwidth of $100Mb/s$ and $1Gb/s$, respectively.

The second observation corroborates the experiments of §7.6 by showing that the latency of both variants of HotStuff vary substantially with the available bandwidth whereas Kauri's are much less affected. Finally, we can also see that HotStuff-bls can outperform HotStuff-secp in certain scenarios but only for a small margin as both systems share the same *sending time* characteristics. To achieve a substantial improvement one needs to drastically reduce the *sending time* through the use of trees, and maximize resource usage through the use of pipelining.

7.9 Heterogeneous Networks

So far we have only considered homogeneous networks, where all links have the same characteristics. We now consider an heterogeneous setup, where participants are deployed in local clusters (with very low RTT and high bandwidth within each cluster) connected over the Internet (with varying bandwidth and RTT between clusters). For this purpose, we adopted the scenario used in the evaluation of ResilientDB [17] in a system with $N = 60$. This allows us to compare the performance of Kauri with the results published for ResilientDB (we did not run ResilientDB ourselves because, at the time of this submission, the public implementation of ResilientDB's GeoBFT protocol was not yet available). When deploying Kauri, we used a tree where the leader is placed in the cluster with the highest bandwidth and lowest RTT to every other cluster (i.e. Oregon) and the internal nodes are located closely to their leaf nodes. Similarly, for the HotStuff variants, the leader is located in Oregon.

Results are depicted in Figure 11. As can be observed, Kauri's throughput substantially outperforms all other systems. This is due to the high RTT, which allows Kauri to pipeline several consensus instances in parallel and hence achieve high throughput. In terms of latency, both HotStuff variants outperform Kauri because the system size is small and hence bandwidth does not become a bottleneck. However, HotStuff latency would quickly grow with the system - as bandwidth becomes the bottleneck, while Kauri's latency would remain largely unaffected. Nonetheless, we stress that Kauri achieves $\approx 10x$ higher throughput than HotStuff variants with only a $\approx 2x$ increase in latency. Interestingly, the performance of Kauri-np is the worse of all systems. This is because without pipelining the high RTT negatively affects the *remaining time*, which ultimately limits throughput.

Note that the Hotstuff-secp throughput results we obtained are substantially lower than those reported in ResilientDB. This is because ResilientDB's authors [17] run

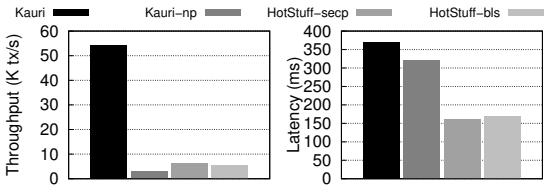


Figure 11. Throughput in the network of [17] with $N = 60$.

N independent HotStuff instances in parallel (where each process is the leader of one such instance) and account for the overall throughput across all instances, which in practice equates to running N independent blockchains. We opted instead to consider the throughput of a single instance deployed in the best possible scenario. In this scenario, Kauri's throughput is on par with the reported throughput of ResilientDB, which is $\approx 55ktx/s$ [17]. However, ResilientDB performance is dictated by the number of clusters while Kauri is independent of this factor. Moreover, ResilientDB tolerates only $f \leq \lfloor \frac{C-1}{3} \rfloor$, where C is the size of the smallest cluster (§1), while Kauri tolerates $f \leq \lfloor \frac{N-1}{3} \rfloor$ faults as classical BFT.

It is worth noting that, in heterogeneous networks, all these protocols require some amount of manual configuration for optimal performance (for instance, selecting the best place for the leader and for internal nodes). Designing methods to automatically find the best deployment configuration for Kauri is left for future work.

7.10 Reconfiguration

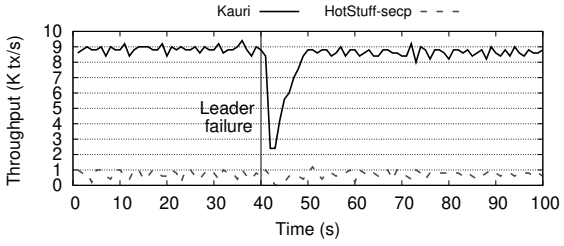
Finally, we evaluate the reconfiguration time in the following faulty scenarios: i) one faulty leader, ii) three consecutive faulty leaders, and iii) f faulty interior or leader nodes. In the first two cases, $f < m$, thus Kauri keeps the tree topology, while in the third case, f is large and hence Kauri needs to fall back to a star. We empirically calibrate the fault detection timeout, by starting with a large value and gradually decreasing it until we found that further decreases would lead to spurious reconfigurations in a stable system. This resulted in a timeout of $0.35s$ for Kauri and $1.7s$ for HotStuff-secp. This difference is explained by the fact that Kauri message dissemination is more regular than HotStuff, due to pipelining, and hence the fault detection timeout can be set more aggressively. Figure 12 presents the results for a system with $N = 100$ in the *global* scenario. We execute each system for 100 seconds (warm-up not shown), inject the fault at 40 seconds and measure the impact on the throughput. As we can observe, for a small number of faults both systems recover to the throughput levels before the fault in ≈ 5 seconds for one faulty leader (Figure 12a) and ≈ 12 seconds for three consecutive faulty leaders (Figure 12b). As shown, Kauri recovers in the same time-frame as HotStuff.

To assess the behavior of Kauri with a large number of faulty nodes (i.e. f) we set up two different scenarios depicted in Figure 12c. In the first scenario (plot line labeled "Kauri leaders") f leaders fail in succession. As expected after f reconfigurations Kauri has fallen back to a star and stabilizes at around the same throughput of HotStuff. In the second scenario (plot line labeled "Kauri internal+leaders"), we start by failing internal nodes of m consecutive trees such that no consensus can be reached for any of these trees. This forces Kauri to degrade to a star after m reconfigurations. Then, we selected faulty processes to serve as leaders for the first f star networks. This is possible because a faulty process can prevent consensus from being reached in two different configurations: if it is an internal node in a tree and if it is subsequently picked as root node for a star. This scenario constitutes the worst-case scenario for Kauri, causing $f+m+1$ reconfigurations, as stated in Section 5.3. In this scenario, the total recovery time is long because all configurations (except the last) timeout. In detail, the timeout is initially set to $1.7s$, doubled after each of the first two reconfigurations, and subsequently capped at $10s$. Since $f = 33$ and $m = 10$, with HotStuff we need a total of $1.7 + 3.4 + 6.8 + 10 \cdot 30 = 311.9s$ to recover and with Kauri we need $10 \cdot 10 = 100s$ more. Nonetheless, after the system stabilizes the performance of Kauri is at the same level as HotStuff.

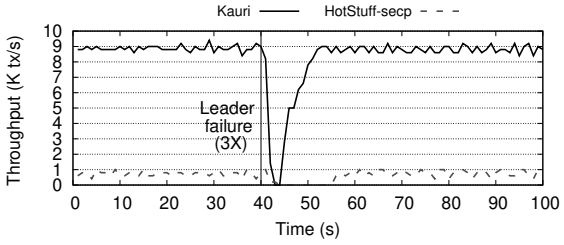
8 Conclusions and Future Work

State-of-the-art permissioned blockchains suffer from important limitations: bottlenecks resulting from work concentration on the leader, throughput decrease when implementing load distribution, or degraded resilience. Kauri overcomes these limitations by introducing a novel pipelining scheme that makes full use of the parallelization opportunities provided by dissemination/ aggregation trees. Furthermore, Kauri uses a reconfiguration strategy that preserves the tree when the number of faults is smaller than the fanout, while still ensuring that a robust configuration is found in a linear number of steps for any number of faults $f < N/3$. In contrast with solutions based on committees, Kauri does not compromise the resilience nor the finality of consensus. Kauri's throughput substantially outperforms HotStuff's in all considered scenarios, reaching up to 28x with only a modest increase in latency. In bandwidth-constrained scenarios Kauri outperforms HotStuff in both throughput and latency. The current implementation of Kauri requires the topology of the tree and the value of the pipelining stretch to be manually configured, using the performance model provided in this paper. We are currently working on algorithms to automate and optimize these configurations.

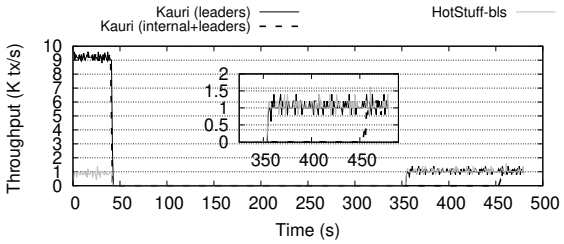
Acknowledgements: We thank our shepherd, Andreas Haeberlen, and the anonymous reviewers for their help in improving the final version of the manuscript. This work was partially supported by CAPES - Brazil (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) and by Fundação para a Ciência e Tecnologia (FCT) under project UIDB/50021/2020 and grant 2020.05270.BD, and via project COSMOS (via the OE with ref.



(a) One faulty leader.



(b) Three consecutive faulty leaders.



(c) f faulty nodes.

Figure 12. Reconfiguration after failure.

PTDC/EEI-COM/29271/2017 and via the "Programa Operacional Regional de Lisboa na sua componente FEDER" with ref. Lisboa-01-0145-FEDER-029271) and project Angainor with reference LISBOA-01-0145-FEDER-031456.

References

[1] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. 2010. Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. *IEEE TDSC* 7, 1 (2010), 80–93.

[2] J. Bernstein, T. Lange, et al. 2013. SafeCurves: choosing safe curves for elliptic-curve cryptography. http://safecurves.cr.yp.to

[3] A. Bessani, J. Sousa, and E. . Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *DSN*. Atlanta (GA), USA.

[4] D. Boneh, B. Lynn, and H. Shacham. 2001. Short signatures from the Weil pairing. Asiacrypt'2001. LNCS 2248.

[5] C. Cachin. 2016. Architecture of the Hyperledger Blockchain Fabric. In *DCCL*. Chicago (IL), USA.

[6] C. Cachin, R. Guerraoui, and L. Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (2nd ed.). Springer.

[7] M. Castro and B. Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI*. New Orleans (LA), USA, 173–186.

[8] B. Cohen and K. Pietrzak. 2020. The chia network blockchain. https://www.chia.net/assets/ChiaGreenPaper.pdf

[9] M. del Castillo. 2020. Forbes Blockchain 50. https://www.forbes.com/sites/michaeldelcastillo/2020/02/19/blockchain-50/?sh=77ed26207553

[10] C. Dwork, N. Lynch, and L. Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (April 1988), 288–323.

[11] M. Eischer and T. Distler. 2018. Latency-Aware Leader Selection for Geo-Replicated Byzantine Fault-Tolerant Systems. In *DSN-W*. Luxembourg City, Luxembourg, 140–145.

[12] D. Balouek et al. 2013. Adding Virtualization Capabilities to the Grid'5000 Testbed. In *Cloud Computing and Services Science*, I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan (Eds.). Communications in Computer and Information Science, Vol. 367. Springer, 3–20.

[13] Z. Amsden et al. 2020. The Diem Blockchain. https://developers.diem.com/docs/technical-papers/the-diem-blockchain-paper/.

[14] M. Fischer, N. Lynch, and M. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382.

[15] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *SOSP*. Shanghai, China, 51–68.

[16] G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *DSN*. Portland (OR), USA.

[17] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proc. VLDB Endow.* 13, 6 (Feb. 2020), 868–883.

[18] S. Hemminger. 2005. Network Emulation with NetEm. In *Proc. of the 6th Australia's National Linux Conference (LCA2005)*. Canberra, Australia.

[19] E. Kokoris-Kogias. 2019. *Robust and scalable consensus for sharded distributed ledgers*. Technical Report. Tech. rep., Cryptology ePrint Archive, Report 2019/676.

[20] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. 2016. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *USENIX Security*. Austin (TX), USA, 279–296.

[21] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *S&P*. San Francisco (CA), USA, 583–598.

[22] L. Lamport, R. Shostak, and M. Pease. 1982. The Byzantine Generals Problem. *TOPLAS* 4, 3 (July 1982), 382–401.

[23] W. Li, C. Feng, L. Zhang, H. Xu, B. Cao, and M. Imran. 2021. A Scalable Multi-Layer PBFT Consensus for Blockchain. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2021), 1146–1160.

[24] D. Mazieres. 2015. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation* 32 (2015).

[25] R. Neiheiser, D. Presser, L. Rech, M. Bravo, L. Rodrigues, and M. Correia. 2018. Fireplug: Flexible and robust N-version geo-replication of graph databases. In *ICOIN*. Chiang Mai, Thailand, 110–115.

[26] R3. Corda Platform. https://www.r3.com/corda-platform/.

[27] T. Ristenpart and S. Yilek. 2007. The Power of Proofs-of-Possession: Securing Multiparty Signatures against Rogue-Key Attacks. In *EUROCRYPT*, M. Naor (Ed.). Springer, 228–245.

[28] C. Stathakopoulou, T. David, and M. Vukolic. 2019. Mir-BFT: High-Throughput BFT for Blockchains. *CoRR* abs/1906.05552 (2019). arxiv1906.05552 http://arxiv.org/abs/1906.05552

[29] G. Veronese, M. Correia, A. Bessani, and L. Lung. 2009. Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *SRDS*. Niagara Falls (NY), USA.

[30] G. Veronese, M. Correia, A. Bessani, and L. Lung. 2010. EBAWA: Efficient Byzantine Agreement for Wide-Area Networks. In *HASE*. San Jose (CA), USA, 10–19.

[31] M. Vukolić. 2015. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In *iNetSec*. Zurich, Switzerland.

[32] P. Wuille. 2018. libsecp256k1. https://github.com/bitcoin/secp256k1

[33] M. Yin, D. Malkhi, M. Reiter, G. Gueta, and I. Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *PODC*. Toronto (ON), Canada, 347–356.