Check for updates

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

RAY NEIHEISER, DAS, UFSC, Florianopolis, Brazil and Pietrzak Group, ISTA - Institute of Science and Technology Austria, Klosterneuburg, Austria
MIGUEL MATOS, INESC-ID, Universidade de Lisboa Instituto Superior Tecnico, Lisboa, Portugal
LUIS RODRIGUES, INESC-ID, Universidade de Lisboa Instituto Superior Tecnico, Lisboa, Portugal

With the growing interest in blockchains, permissioned approaches to consensus have received increasing attention. Unfortunately, the BFT consensus algorithms that are the backbone of most of these blockchains scale poorly and offer limited throughput. In fact, many state-of-the-art BFT consensus 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 limits the throughput for deeper trees. In this paper we propose Kauri, a BFT communication abstraction that sustains high throughput as the system size grows by leveraging a novel pipelining technique to perform scalable dissemination and aggregation on trees. Furthermore, when the number of faults is moderate (arguably the most common case in practice), our construction is able to recover from faults in an optimal number of reconfiguration steps. We implemented and experimentally evaluated Kauri with up to 800 processes. Our results show that Kauri outperforms the throughput of state-of-the-art permissioned blockchain protocols, by up to 58x without compromising latency. Interestingly, in some cases, the parallelization provided by Kauri can also decrease the latency.

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

Additional Key Words and Phrases: Distributed Systems, Byzantine Fault Tolerance, Blockchain, Vote Aggregation, Pipelining

1 Introduction

The increasing popularity of blockchains in addressing an expanding set of use cases, from enterprise to governmental applications [15], led to a growing interest in permissioned blockchains, such as Hyperledger Fabric [2]. In contrast to their permissionless counterparts, permissioned blockchains can ensure deterministic transaction finality, which is a key requirement in many settings [38]. Most permissioned blockchains are based on variants of classical Byzantine fault-tolerant (BFT) consensus algorithms, that can offer high throughput in small systems [40], but scale poorly with the number of participants [16, 25]. This is a strong limitation, given that emerging use cases for permissioned blockchains require the system to scale to hundreds of participants [20]. For instance, a recent paper from IBM Research [34] discusses the need to extend HyperLedger to support deployments above 100 nodes.

The scalability limitations of previous protocols stem from both bandwidth and processing bottlenecks resulting from the large number of messages that need to be sent, received, and processed to reach consensus. For instance, the well-known PBFT protocol [12] organizes participants in a clique topology and uses an all-to-all

Authors' Contact Information: Ray Neiheiser, DAS, UFSC, Florianopolis, Santa Catarina, Brazil and Pietrzak Group, ISTA - Institute of Science and Technology Austria, Klosterneuburg, Lower Austria, Austria; e-mail: ray.neiheiser@ist.ac.at; Miguel Matos, INESC-ID, Universidade de Lisboa Instituto Superior Tecnico, Lisboa, Lisboa, Portugal; e-mail: miguel.marques.matos@tecnico.ulisboa.pt; Luis Rodrigues, INESC-ID, Universidade de Lisboa Instituto Superior Tecnico, Lisboa, Lisboa, Portugal; e-mail: ler@tecnico.ulisboa.pt.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 1557-7333/2025/9-ART
https://doi.org/10.1145/3769423

Just Accepted

communication pattern that incurs a quadratic message complexity. Although there have been many proposals to extend and improve several aspects of PBFT (such as [9, 17, 36, 37]), most preserve its communication pattern.

Hotstuff [40] avoids the quadratic communication cost by having a single leader aggregate all votes and then distribute them to the remaining processes. However, this creates a performance bottleneck at the leader: first, the leader has to send the (potentially large) block to all processes, where the bandwidth consumption grows linearly with the number of processes; second, the leader has to verify all $N - 1$ signatures.

Meanwhile, DAG-based protocols such as Narwhal [14] avoid the bandwidth bottleneck at the leader by decoupling block dissemination from block ordering. In this category of protocols, first all processes broadcast blocks using a DAG-based mempool protocol and, later, consensus is primarily used for finality. However, all processes still need to verify $N - 1$ signatures per round and, as such, there is still a processing bottleneck.

Existing tree-based approaches, such as Byzcoin [22] and Motor [23], address both the computational and bandwidth bottlenecks. However, these approaches suffer from two key drawbacks. First, they fail to achieve high throughput due to the inherent latency increase associated with tree-based dissemination. Second, they lack efficient mechanisms to deal with Byzantine nodes in the tree that can disrupt the operation of the tree leading to long reconfiguration periods and eventually falling back to a star or clique configuration.

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 degradation of the tree topology to a star or clique 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 depth) 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 the use of tree-based dissemination/aggregation and pipelining is that using arbitrary pipeline values results in poor performance: under-pipelining (i.e. pipelining less than the system capacity allows) fails to take advantage of the available parallelization opportunities, while over-pipelining (i.e. starting more instances than the system can handle) 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 maximizes performance.

To overcome the slow reconfiguration of existing tree-based approaches, we introduce a novel reconfiguration strategy that, when the number of consecutive faults is moderate (arguably the most common scenario), enables Kauri to quickly reach a configuration where all internal nodes are correct. We refer to such a configuration as *robust*.

More precisely, for a tree fanout of m, if $f \leq f_r < \frac{N-1}{3} * \frac{m^2}{N-1+m^2-m} + 1$, Kauri is guaranteed to find a robust tree-based configuration in $f + 1$ reconfiguration steps, which is optimal. Furthermore, as demonstrated in §6, even in the presence of a large number of failures, our reconfiguration strategy has a high probability of reaching a robust tree within $f + 1$ steps. Consequently, Kauri only needs to fall back to a star topology in scenarios where $f > f_r$ consecutive faults occur and no robust configuration can be found within $f + 1$ steps.

Kauri offers the same fault-tolerance of traditional approaches (i.e. ensures safety as long as $f \leq \frac{N-1}{3}$), and distributes the computational and bandwidth load among internal nodes allowing it to scale with the number of processes and achieve high throughput. We implemented Kauri and evaluated it under different realistic scenarios with up to 800 processes. Our results show that Kauri outperforms HotStuff's throughput by up to 58x and in some scenarios it offers better latency. When compared with Narwhal, Kauri achieves on average 100x better latency while Narwhal offers on average 3x better throughput. However, as the system size increases, due to

Just Accepted

Narwhal's bandwidth and processing bottlenecks, we observe that when the system size reaches approximately 300 nodes, Kauri also starts to offer better throughput.

In short, the paper makes the following contributions:

- We present a set of abstractions that support the use of aggregation/dissemination trees in the context of Byzantine consensus algorithms;
- We introduce a performance model that shows how pipelining can be used to fully leverage the parallelisation opportunities offered by the use of trees;
- We precisely outline the necessary and sufficient conditions that allow efficient reconfiguration of trees without having to fall back to a star topology;
- We present Kauri, the first tree-based communication abstraction for BFT consensus protocols that achieves higher throughput than HotStuff and lower latency than Narwhal. In certain scenarios, particularly at scale, Kauri achieves better throughput and latency than either system;
- We present an extensive experimental evaluation of Kauri in realistic scenarios with up to 800 nodes.

The rest of this paper is organized as follows: §2 discusses related work and how it compares to Kauri. §3 introduces the system model which follows the same assumptions as classical BFT consensus algorithms such as PBFT. §4 presents the tree construction algorithms, §5 introduces the pipelining techniques and the respective performance model. §6 details the reconfiguration algorithms, that allow the system to be reconfigured in a small number of steps in the common case. §7 discusses the implementation details, and §8 evaluates and compares Kauri with other state-of-the art approaches. Finally, §9 discusses the limitations of our work, and §10 concludes the paper.

2 Related Work

There is a wide selection of Byzantine fault-tolerant consensus algorithms that were designed with the goal of improving PBFT [12] scalability bottlenecks. HotStuff [40] avoids a quadratic message complexity by having only the leader send/collect messages directly to/from all other processes, i.e. communication is based on a star topology centered at the leader. This approach results in a linear message complexity, but the leader is still required to receive and validate votes from at least $N - f$ other processes. At the time of writing, the publicly available implementation of HotStuff uses *secp256k1* [39] signatures, a highly efficient elliptic curve algorithm that is also used in Bitcoin [8]. In this implementation, the leader is required to relay the full set of signatures to all processes. Alternatively, it is possible to use multisignatures, such as BLS [10], to reduce the message size at the expense of a significant increase in the computational load at the leader. In HotStuff, each protocol round takes two communication steps. To mitigate the negative effect that the additional round latency can have on throughput, HotStuff uses pipelining: 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 piggybacking information from multiple consensus instances in a single message and is critical for reducing latency and improving throughput. Unfortunately, in a star topology, pipelining increases the burden on the leader, further amplifying the scalability limitations that stem from processing and bandwidth bottlenecks. As a result, and as we show in the evaluation (§8), HotStuff is inherently non-scalable as the system performance is limited by the processing and bandwidth capacity of the leader.

MirBFT [34] solves the bandwidth bottleneck by using concurrent leaders that propose in parallel. However, if one of the leaders is faulty a view change is necessary. As such, even a small number of byzantine failures can force the system to change its view until a configuration is reached without a faulty leader. This might require a factorial number of reconfigurations which can have a substantial negative impact on performance. A followup work called ISS [35] improves on this by using parallel instances of sequenced broadcast. However, this still

Just Accepted

results in a computational bottleneck where for each round at least one node has to verify the signatures of all other nodes, presenting an inherent scalability limitation.

A way to circumvent the scalability constraints is to select a small committee such as in Algorand [19]. However, this approach reduces the resilience of the system, as safety is now probabilistic in function of the committee size and not of the entire system size. Alternatively, systems such as Steward [1], Fireplug [29], ResilientDB [31] or Multi-Layer PBFT [27] organize processes in hierarchical groups to achieve low message complexity and balance the bandwidth load. However, these approaches sacrifice resilience by tolerating significantly less than $f = \frac{N-1}{3}$ failures.

Approaches such as Byzcoin [22], Motor [23], and Omniledger [24] 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 messages from the leader to the other processes, and to aggregate votes using cryptographic primitives such as multisignatures [10, 22]. The use of trees reduces the number of messages any single process has to send, receive, and process (which 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 increase in the latency of each consensus round. In fact, while in PBFT a round can be executed within a single communication step, and in HotStuff it requires two communication steps (i.e., one roundtrip), trees require $2d$ communication steps, given the depth of the tree d. If the blockchain protocol only starts a new consensus instance after the previous one terminates, this per-round latency increase has a direct negative impact on the system throughput. In particular, the performance advantages that stem from the load distribution provided by the tree can be easily 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 due to the increased number of communication steps required to complete each round. As we show in the evaluation (§8) this results in low throughput.

Another disadvantage of trees is their fragility to faults and the resulting potentially long reconfiguration procedures. 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 correct. Moreover, if the leader is faulty, the system is guaranteed to recover after $f + 1$ reconfiguration steps (also known as view changes in the literature). When using trees, progress is guaranteed if all internal nodes of the tree are correct (this is a sufficient but not necessary condition as discussed in §4). However, finding a configuration without faulty internal nodes has combinatorial complexity [23]. Due to these challenges, Byzcoin [22] quickly falls back to a clique when faults occur. Motor [23] and Omniledger [24] 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, and considering 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, note that this strategy was only designed for trees with a maximum depth of $d = 2$.

Next, the Chain Protocol [4] constructs a chain of N nodes which allows using MACs for easy and scalable authentication. However, not only does the latency increase in function of the system size N, but the protocol does not tolerate failures – a single fault forces the system to fallback to less performant variants.

Recently, DAG-based consensus protocols such as Narwhal and Tusk [6, 14] have addressed the bandwidth bottleneck at the leader by decoupling block dissemination from block ordering. Unlike traditional chain-based structures, these protocols allow processes to broadcast blocks concurrently, resulting in a directed acyclic graph (DAG) of blocks rather than a linear chain. Subsequently, consensus is applied only to determine the order and finality of blocks. However, since the protocol relies on broadcast communication, each node must verify the

Just Accepted

Table 1. Comparison of existing BFT consensus algorithms.

System	Bandwidth balancing	CPU balancing	$N = 3f + 1$ resilience	Latency compensation	Quick recovery
PBFT [12]	✗	✗	✓	✗	✓
HotStuff [40]	✗	✗	✓	✗	✓
Ebawa [37]	✗	✗	✓	✓	✓
Steward [1]	✓	✓	✗	✗	✗
Fireplug [29]	✓	✓	✗	✗	✗
ResilientDB [31]	✓	✓	✗	✗	✗
Multi-Layer [27]	✓	✓	✗	✗	✓
Algorand [19]	✓	✓	✓	✗	✗
Byzcoin [22]	✓	✓	✓	✗	✗
Omniledger [24]	✓	✓	✗	✗	✗
Chain [35]	✓	✓	✓	✗	✗
Narwhal [14]	✓	✗	✓	✓	✓
ISS [35]	✓	✗	✓	✗	✓
Kauri (this work)	✓	✓	✓	✓	✓

signatures of all N blocks received per round, creating a computational bottleneck. Additionally, because processes broadcast blocks independently, the same transaction may appear in multiple blocks, leading to redundant entries in the blockchain. Although such duplicates are eventually discarded, they still incur unnecessary communication and storage overhead.

Table 1 summarizes our discussion of the systems based on the criteria discussed above. The first and second column captures if an approach distributes the bandwidth and computational load among a set of processes, respectively. The third column indicates if a given approach displays optimal resilience (i.e., $N = 3f + 1$). The fourth column indicates if the system can provide high throughput independent of the network latency. Finally, the last column indicates if the approach can recover deterministically in a linear number of configuration steps. 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.

3 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 $\{c_1, c_2, \ldots, c_c\}$. We also assume the existence of a Public Key Infrastructure used by processes to distribute the keys required for authentication, message signing, and verification. 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 [32]. We assume the Byzantine fault model, where at most $f \leq \frac{N-1}{3}$ 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.

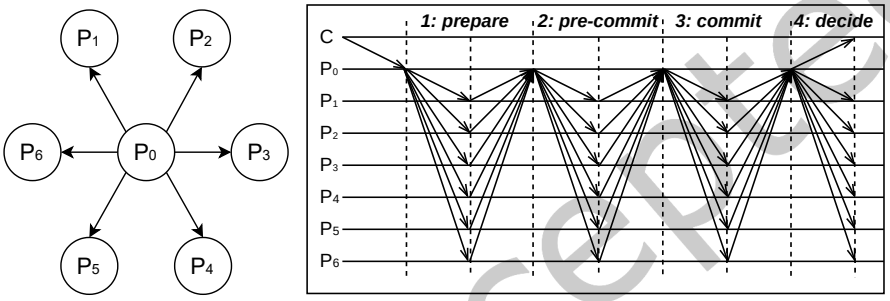
Processes communicate via perfect point-to-point channels, whose interface includes a SEND primitive (used to send a value) and a DELIVER (used to notify a recipient that a value has been received). Perfect point-to-point channels have the following properties:

- *Validity:* If a process p_j delivers a value v on a channel over an edge e_{ij}, v was sent by p_i.
- *Termination:* If both p_i and p_j are correct, if p_i invokes SEND then eventually p_j DELIVERS v.

Just Accepted

Notation	Description
N	Total number of processes in the system
f	Maximum number of tolerated faults. By assumption $f = \frac{N-1}{3}$
m	Tree fanout
d	Tree depth
f_a	Number of actual faults, subject to $f_a \leq f$
f_r	Maximum number of faults for robust tree construction
B	Block size
b	Link bandwidth

Table 2. Notation used throughout the paper.



(a) Topology (b) Communication pattern

Fig. 1. HotStuff topology and communication pattern in a system with N=7 processes.

These are implemented using mechanisms for message re-transmission and detection and suppression of duplicates [11]. To circumvent the impossibility of consensus [18], we assume the partial synchrony model [16]. 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. Similarly to other works, safety is always preserved and the partial synchrony assumptions are necessary only to ensure liveness [40].

The notation used throughout the rest of the paper is summarized in Table 2.

4 Design of Dissemination/Aggregation Trees

Instead of designing a completely new consensus algorithm from scratch, we focus on the communication primitives that are needed to reach consensus. The key idea is to replace the dissemination and aggregation patterns used by systems such as HotStuff, which is based on a star topology, by our new pattern 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.

4.1 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

Just Accepted

different implementations. HotStuff reaches consensus in four communication rounds as illustrated in Figure 1 for a system with seven processes. 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 Byzantine quorum of processes. 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 processes have validated and accepted the block proposed by the leader.

Second round: In the dissemination phase, the leader broadcasts the *prepare* quorum previously collected. 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 collected in the second round. 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.

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

To implement these rounds, HotStuff uses the following two communication 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. ***Robust Star:*** *A star is considered robust if the leader is correct, and non-robust if the leader is faulty.*

Definition 2. ***Reliable Dissemination:*** *After the GST, and in a robust star, all correct processes deliver the data sent by the leader.*

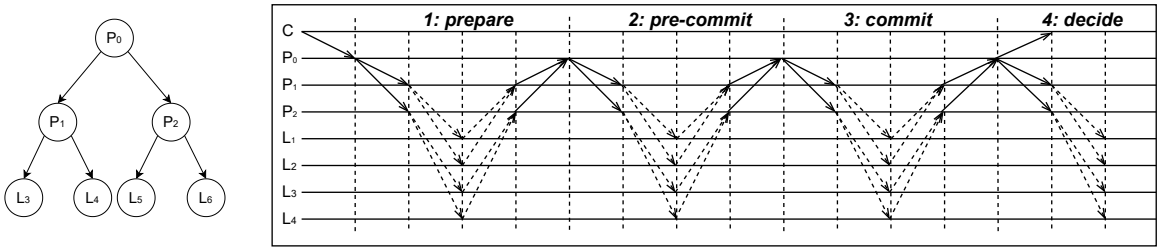
Definition 3. ***Fulfillment:*** *After the GST, and in a robust star, the aggregate collected by the leader includes at least $N - f$ votes.*

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. 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. We refer the reader to the HotStuff paper for the full details [40].

4.2 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 forward it to their children, and so forth. The primitive *waitFor* is implemented by having the leaf processes send their signatures to their parent. The parent then aggregates those signatures with its own and sends the aggregate to their parent. This process is

Just Accepted



(a) Topology (b) Communication pattern

Fig. 2. Tree topology and communication pattern in a system with N=7 processes.

repeated until the final aggregate is computed at the root of the tree. This process is illustrated in Figure 2 for a system with seven processes.

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

Definition 4. ***Safe Edge:*** *An edge is said to be safe if the corresponding vertices are both correct processes.*

Definition 5. ***Robust Tree:*** *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.*

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 (§6). Note that the 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. We capture this in the definition of a *quorum q-robust tree* as follows:

Definition 6. ***Quorum q-Robust Tree:*** *A tree is quorum q-robust iff: i) the leader process is correct and ii) there are at least $q - 1$ correct process that are reachable from the root using safe edges only.*

The use of trees that are *quorum q-robust* with $q = N - f$ is a sufficient and necessary condition to achieve consensus. For simplicity of explanation, in the following sections we provide the intuition and algorithms for our approach based on the assumption of a robust tree. Later, in § 6.3, we introduce the extensions required to achieve consensus on a quorum q-robust tree.

4.3 Dissemination and Aggregation

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

4.3.1 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. For this purpose, we need to use a signature scheme that supports signature aggregation. In the current implementation 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 [10]. The burden imposed on each internal node (including the root) is $O(m)$,

Just Accepted

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 [10].

To make the protocol description independent of the concrete signature scheme used in the implementation, we capture the properties we need from the signature scheme using the notion of 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 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$((p_i, v))$

4.3.2 Impatient Channels. Processes use the tree to communicate. 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, messages may be duplicated (in this case, duplicates need to be discarded), delayed, or may even never be sent (in this case, the recipient may timeout and proceed without waiting for that particular message). We encapsulate the functionality required to discard duplicates and to manage timeouts in an abstraction that we have named *impatient channels*. From the point of view of upper-layers, each invocation of the dissemination/aggregation protocol uses a different impatient channel, that transmits a single message and where the receiver either returns the value sent or $\perp$ (but never delivers duplicate values or messages sent in previous steps of the protocol). Note that single-use impatient channels are introduced just to simplify the specification of the protocol. In practice, multiple single-use channel instances will use the same underlying point-to-point channel. In the implementation this is achieved by using TCP, timeouts, assigning a unique identifier to each instance, and tagging the corresponding message with this identifier. 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 returns 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.

We now have the machinery to implement *broadcastMsg* and *waitFor* primitives which we discuss next.

4.3.3 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

Just Accepted

Algorithm 1 Impatient Channels

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

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
```

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 IC.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. □

4.3.4 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.*

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

Just Accepted

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
```

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 tree, 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 ensuring that the node will unblock and relay all collected signatures from all correct child processes. Therefore Algorithm 3 guarantees Fulfillment. □

4.3.5 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 (§8). 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, in turn, may negatively affect they system throughput. We discuss how to mitigate this in §5.

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. $f + 1$), for instance, by rotating the leader when the leader is suspected. 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 §6 we introduce a reconfiguration strategy that builds robust configurations in a small number of steps.

5 Mitigating Tree Latency

In this section we introduce 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

Just Accepted

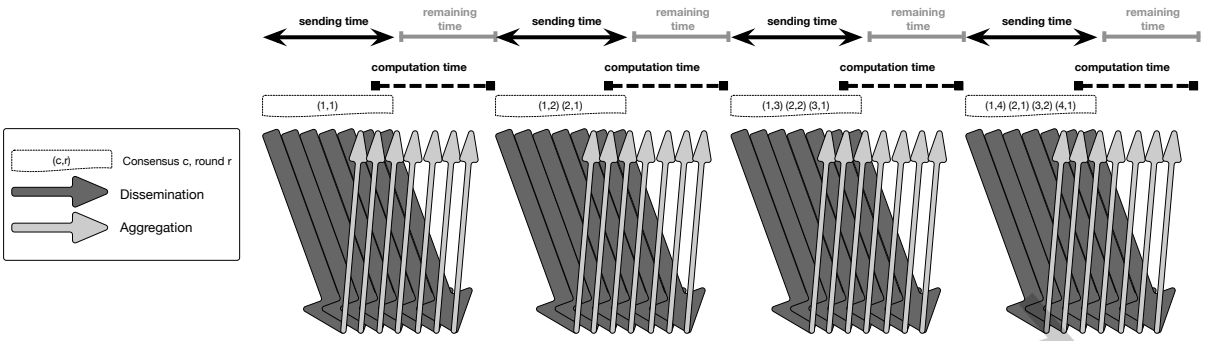


Fig. 3. Pipelining in HotStuff from the perspective of the leader process.

HotStuff. While this, at first look, may appear a fundamental 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 (§8), in certain scenarios, Kauri can achieve not only higher throughput than HotStuff but also lower latency.

5.1 Pipelining in HotStuff

We start by providing an overview of how pipelining is used in HotStuff, with the help of 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 processes. 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 optimistically 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.

5.2 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 processes 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

Just Accepted

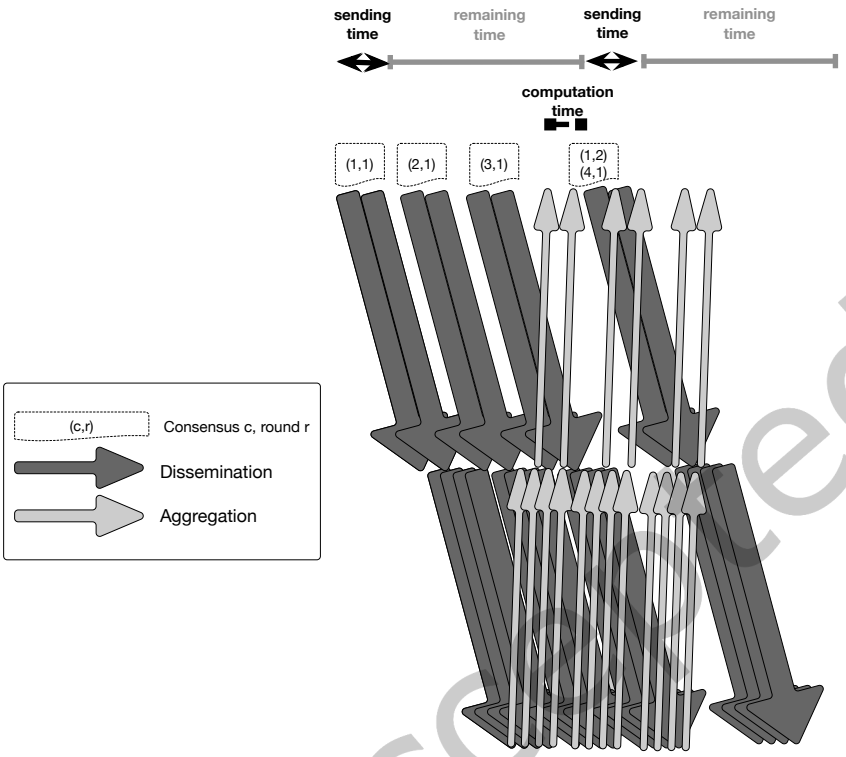


Fig. 4. Pipelining in Kauri.

from consensus instances/rounds that are farther away in the pipeline. Hence, the first quorum for instance 1 is collected upon the arrival of instance 4, the second quorum at instance 7, and the final quorum (required for finality) at instance 10. Once a node receives these three subsequent quorums—culminating with instance 10—it can finalize instance 1, offering the same safety guarantees as HotStuff. This increase in the pipelining depth allows for a higher degree of parallelism, and hence throughput.

5.3 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 §8).

Remaining time: the time that elapses from the 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:

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

Just Accepted

where d is the depth or 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 processes instead of $N-1$. 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 $N = 400$ processes, organized in a tree with fanout $m = 20$, the maximum speedup we can expect Kauri to offer is $\frac{400-1}{20} = 19.95$.

6 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 immediately back to a star topology as soon as a single fault occurs.

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 5). Thus, a reconfiguration strategy that considers all possible configurations may require a factorial number of steps to find a robust tree. Next, we discuss how our reconfiguration strategy addresses this problem.

6.1 Modeling Reconfiguration as an Evolving Graph

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

Definition 7. ***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, in practice, this is not enough because it does not bound the number of reconfigurations until a robust graph is found. Given that 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.*

Just Accepted

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

```
1:  function INIT(N, m)          ▷ Initialize the evolving tree with the set of nodes N and fanout m
2:      B ← ∅                    ▷ Initialize the set of bins
3:      for all i ∈ N do         ▷ For each node
4:          B^(i mod m) ← B^(i mod m) ∪ i   ▷ Assign the node i to one of the m bins
5:      end for
6:  end function
7:  function BUILD(k)
8:      i ← k mod m
9:      G^i ← all possible trees whose internal nodes are drawn exclusively from B^i
10:     T^k ← pick any tree at random from G^i
11:     return T^k
12: end function
```

Definition 8. ***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.*

As we discussed earlier, the use of a q-robust tree with $q = N - f$ is a sufficient but not necessary condition to achieve consensus. In fact, it is enough that the reconfiguration is able to find a quorum q-robust tree, which we capture in the following property:

Definition 9. ***t-Bounded q-Robust Conformity**: a recurringly robust evolving graph $\mathcal{G}$ exhibits t-Bounded q-Robust Conformity if a quorum q-robust static graph appears in $\mathcal{G}$ at least once every t consecutive static graphs.*

6.2 Reconfiguring for t-Bounded Conformity

We now introduce Algorithm 4, that constructs an evolving graph offering t-Bounded Conformity. The algorithm initiates the preliminary structures with a randomly shuffled set of processes which is then equally distributed over all m bins (line 4). Then, it builds an evolving graph by creating trees whose internal nodes are drawn exclusively from a given bin following a round-robin strategy. This is done by picking a sequence number i based on the total number of bins m and the current configuration k. In detail, for a given bin B^i (line 8), it builds a tree T^k by assigning nodes from this bin B^i to all internal nodes of the tree including the root (lines 9 to 11). Following that, it assigns the remaining nodes to random leaf positions in the tree (line 10).

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

Proof. Based on Algorithm 4 we construct m disjoint bins and, as long as at most $f_a < m$ faults occur, at least one of the bins is guaranteed to be composed exclusively of correct nodes. At each consecutive step, the algorithm picks a tree whose internal nodes are drawn exclusively from distinct bins, hence guaranteeing that a robust tree is found within at most m steps. □

A limitation of this 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 created and, therefore, the maximum value for t. In a balanced tree of fanout m we can obtain at most m disjoint bins with enough capacity to assign all the internal nodes in the tree, such that this algorithm allows us to achieve at most $(m-1)$-Bounded Conformity.

6.3 Reconfiguring for t-Bounded q-Robust Conformity

Algorithm 4 introduced before achieves Optimal Reconfiguration only if the number of faults f_a that occur are at most $f_a < m < \frac{N}{3}$, where m is the fanout of the tree. We now present Algorithm 5 that is able to tolerate

Just Accepted

Algorithm 5 Construction of an Evolving Tree with t-Bounded q-Robust Conformity

1: **function** INIT($\mathcal{N}$, m) ▷ Initialize the evolving tree with the set of nodes $\mathcal{N}$ and fanout m
2: $B \leftarrow \emptyset$ ▷ Initialize the set of bins
3: **for all** $i \in \mathcal{N}$ **do** ▷ For each node
4: $B^{i \bmod m} \leftarrow B^{i \bmod m} \cup i$ ▷ Assign the node i to one of the m bins
5: **end for**
6: **end function**
7: **function** BUILD(k)
8: $i \leftarrow k \bmod m$
9: $\mathcal{G}^i \leftarrow$ all possible trees whose internal nodes are drawn exclusively from B^i.
10: $r \leftarrow B^i[\lfloor \frac{k}{m} \rfloor]$ ▷ Pick root from B^i
11: $T^k_r \leftarrow$ pick any tree at random from $\mathcal{G}^i$ with root r
12: **return** T^k
13: **end function**

faulty internal nodes and hence achieves t-Bounded Conformity for much larger f_a. The rationale is similar to Algorithm 4.

We initiate the algorithm by equally distributing the nodes over all bins B^i (line 4). Next, we again pick a sequence number in function of m and k and choose the respective bin B_i. However, instead of iterating over each bin at most once and selecting the root for each iteration at random, the algorithm iterates over each bin multiple times and selects the root node within each bin in a round-robin fashion (line 10). Following this, the remaining processes are again distributed at random to the leaf positions in the tree.

THEOREM 4. *Algorithm 5 constructs an evolving graph that satisfies t-Bounded q-Robust Conformity as long as* $q < \frac{2N}{3}$ *and* $f_a < \frac{N-1}{3} * \frac{m^2}{N-1+m^2-m} + 1 \leq t$

PROOF. We first show that Algorithm 5 satisfies q-Robust Conformity. We prove this by contradiction. For $f_a < m$ we know based on Theorem 3 that we can construct a robust trees in t steps. We therefore only discuss here the case where $m \leq f_a < \frac{N-1}{3} * \frac{m^2}{N-1+m^2-m} + 1$, where there might be one or more faulty nodes in every bin. Assume that no quorum could be collected. As we draw the internal nodes of a given tree exclusively from a single bin, this means that we might have faulty internal nodes. Due to this, consensus may be prevented as faulty internal nodes might block correct nodes (their children) from participating in consensus. Consider the case where the root is correct and one or more internal nodes are faulty. Note that each faulty internal node may prevent at most $\lceil \frac{N-m-1}{m} \rceil$ (all its children) from participating in consensus. As nodes are equally distributed over all m bins, there is at least one bin with at most $\lfloor \frac{f_a}{m} \rfloor$ faulty processes. Thus, in a tree constructed from this bin, at most $\lceil \frac{N-m-1}{m} \rceil * \lfloor \frac{f_a}{m} \rfloor$ processes may be prevented from participating in consensus, in addition to the f_a faulty nodes themselves. Thus, as long as $\lceil \frac{N-m-1}{m} \rceil * \lfloor \frac{f_a}{m} \rfloor + f_a \leq \frac{N-1}{3}$ we can collect a quorum and solve consensus, which holds under $f_a < \frac{N-1}{3} * \frac{m^2}{N-1+m^2-m} + 1$. In the case the root is faulty, Algorithm 5 constructs trees by rotating, for each bin, the root node in a round-robin fashion, eventually guaranteeing a correct process at the root. In either case, the algorithm can collect a quorum and solve consensus, which leads to a contradiction.

We now show that Algorithm 5 achieves t-Bounded Conformity for $f_a < t$. We prove this by contradiction. Assume that Algorithm 5 could not find a q-robust configuration in $f_a + 1$ steps but, would instead, require at least $f_a + 2$ steps. As shown above, Algorithm 5 eventually reaches a robust configuration. As nodes are equally distributed over all m bins, there is at least one bin with at most $\lfloor \frac{f_a}{m} \rfloor$ faulty processes. In the worst case, this is the last bin, i.e. bin m. As such, Algorithm 5 iterates over all m bins in sequence and rotates the root until

Just Accepted

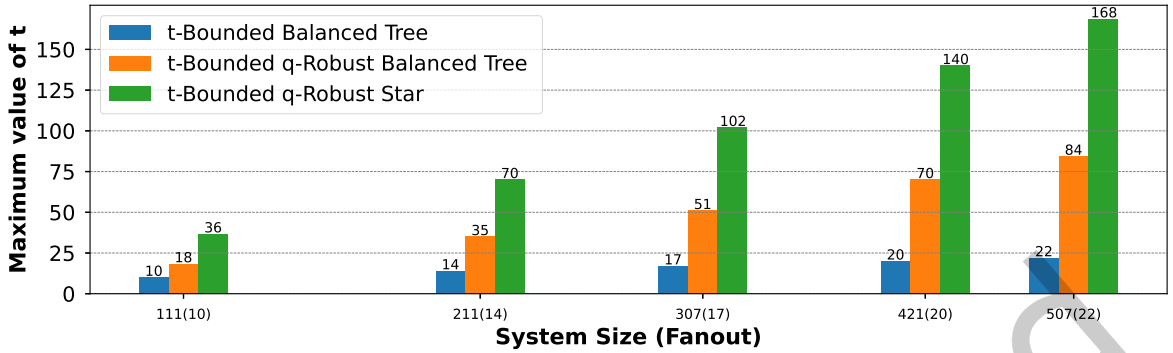


Fig. 5. Maximum value for t in a perfectly balanced tree vs t in a star topology, for varying system sizes.

a correct configuration is reached which requires at most $\lfloor \frac{f_a}{m} \rfloor * m + 1$ steps. This leads to a contradiction as $\lfloor \frac{f_a}{m} \rfloor * m + 1$ is no bigger than $f_a + 1$. □

Algorithm 5 achieves t-Bounded q-Robust Conformity for $f_a < \frac{N-1}{3} * \frac{m^2}{N-1+m^2-m} + 1 \leq t$. Naturally, in adversarial scenarios, the number of actual faults f_a can approximate the maximum number of faults $f = \frac{N-1}{3}$ tolerated by the system. In such scenarios, Kauri may need to degrade the tree to a star topology.

Figure 5 illustrates the maximum value of t that we can expect to achieve when aiming at t-Bounded and t-Bounded q-Robust in a tree when compared with the optimal case that can be achieved when using a star topology, where the maximum value for t is $\frac{N-1}{3}$). For simplicity, we show a perfectly balanced tree with depth $d = 2$, and varying system sizes, with the fanout m adjusted accordingly. As it is possible to observe, Algorithm 5 is able to reach a robust configuration for much higher values of t than Algorithm 4. The robustness of the tree increases proportionally to the system size and fanout, which is appealing since we expect large system sizes to be subject to more faults. In fact, the extended reconfiguration algorithm Kauri can tolerate an actual number of faults f_a that is roughly half of the optimal value $\frac{N-1}{3}$ before falling back to a star topology.

6.4 Reconfiguration in Practice

From Theorem 4, Algorithm 5 only guarantees the construction of a t-Bounded q-Robust tree in $f_a + 1$ steps, when $f_a < \frac{N-1}{3} * \frac{m^2}{N-1+m^2-m} + 1$. We denote this upper bound of failures as f_r. Thus, when running Algorithm 5, if after $f_r + 1$ reconfiguration steps a quorum $q-$robust tree is not found, this means that the system is facing an actual number of faults $f_a > f_r$ and, therefore, a robust configuration is only guaranteed to be found if the system falls back to a star topology.

However, in certain scenarios, Algorithm 5 may be able to find a robust configuration in a number of reconfiguration steps less or equal to $f_r + 1$ even when $f_a > f_r$. In fact, as long as there is a bin with at most $\lfloor \frac{f_r}{m} \rfloor$ faulty nodes, Algorithm 5 can find a robust configuration (see Theorem 4) and thus, it only fails to find a robust tree in $f_r + 1$ steps when the $f_a > f_r$ faulty nodes are evenly distributed among all bins. As we discuss next, if faults are randomly distributed among the bins and leaves, this scenario is unlikely to occur.

We can model the probability of having a bin with at most $\lfloor \frac{f_r}{m} \rfloor$ nodes as an Urn problem [3]. In detail, there are N nodes, out of which f are faulty and $N - f$ are correct. For m rounds (the number of bins) we draw $\frac{N}{m}$ nodes (the number of nodes in each bin) without replacement and calculate the probability of picking at most

Just Accepted

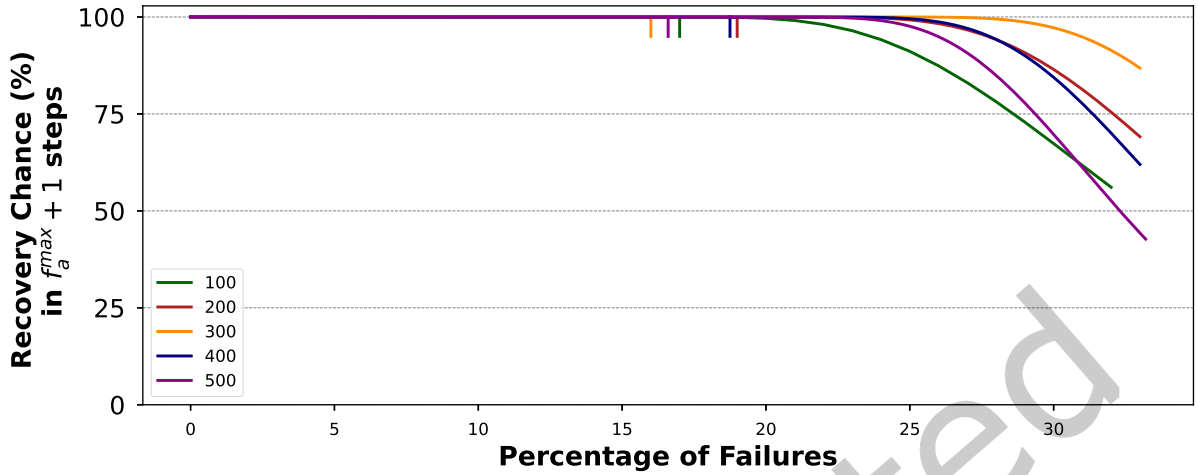


Fig. 6. Probability of finding a quorum $q-robust$ tree for an increasing number of faults. The vertical bars for each system size delimit f_r up to which Algorithm 5 ensures the construction of such a tree.

$\lfloor \frac{f_r}{m} \rfloor$ faulty nodes (the number of faulty internal nodes we can tolerate and still achieve consensus following Theorem 4). In this context, each reconfiguration attempt follows a hypergeometric distribution [30]. If we sum up the probability of all successful branches in a probability tree based on the above scenario, this translates to the estimated probability of finding a quorum q-robust tree for any f_a.

Figure 6 depicts the probability of finding a t-Bounded q-Robust tree in $f_r + 1$ reconfiguration steps in several scenarios, considering systems with different sizes N (100, 200, 300, 400, 500), fanouts m (10, 14, 17, 20, 22), and experiencing different numbers of faults. In the plot, we represent the number of faults as a percentage of the system size. Interestingly, our proposed algorithm has a very high chance to find a robust tree in $f_r + 1$ reconfiguration steps, independently of the system size up to $\frac{N}{4}$. Even at $f_a = \frac{N-1}{3}$ failures, for most configurations, Algorithm 5 offers a probability higher than 50% of finding a robust tree in $f_r + 1$ steps.

6.5 Conservative Reconfiguration and Graceful Degradation

In our experiments, we have adopted the following pragmatic approach to reconfiguration: we execute Algorithm 5 for $f_r + 1$ steps and then, if no robust tree is found, Kauri falls back to a star (and to the performance of HotStuff). Thus, in the worst case scenario, Kauri performs $f_r + \frac{N}{3} + 1$ reconfigurations until a star topology with a non-faulty leader is found. When the number of actual faults is not large, i.e., when $f_a \leq f_r$, Kauri is guaranteed to recover in at most $f_a + 1$ steps, and avoids falling back to a star. Furthermore, as shown in the previous section, if the adversary is unable to target nodes freely, and faults are distributed at random among bins, this conservative approach may still avoid falling back to a star even when $f_a > f_r$.

It would be possible to consider other more optimistic reconfiguration approaches, that would also take longer to recover in the worst case, but that would avoid falling back to a star in a wider range of cases. This could be achieved by running Algorithm 5 for a number of steps larger than $f_r + 1$. The analysis presented in the previous section indicates that such an approach would be viable. However, we let the study of such approaches for future work.

Just Accepted

7 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 have also added support for the BLS cryptographic scheme by including the publicly available implementation used in the Chia Blockchain [10, 13]. 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.

Since 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 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 §5 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. We leave for future work how this value could be automatically adapted at runtime.

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 [40]. In turn, the candidate leader waits for $N - f$ *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 $N - f$ *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.

We define the following timeouts in the system. The limit on message transmissions between two processes δ after which a process no longer waits for the reception and the absolute upper limit Δ (as described in § 3). Based on the value of δ we calculate the *round termination timeout* as $RoundTimeout = \delta \cdot d \cdot 2$, where d is the tree depth, after which a reconfiguration is enacted. Every time a reconfiguration is enacted, δ is doubled until reaching or exceeding Δ at which it is capped.

Signature Aggregation and Verification: To process signatures from their children, a parent node can follow two strategies: verify-then-aggregate and aggregate-then-verify.

In the verify-then-aggregate approach, the parent first verifies the signatures from each children and then aggregates them together. This results in each internal node having to perform m verifications (assuming a perfectly balanced tree) and a single aggregation. Since verification is more expensive than aggregation, this strategy results in a higher computational than the other approach.

In the aggregate-then-verify approach, instead of having each process verify each incoming signature and then constructing the aggregate, we directly aggregate the signatures and only verify the aggregate afterwards. This reduces the computational load significantly as only a single verification is performed rather than a verification for each children. If a child process contributes an invalid signature, the verification of the aggregate fails and all signatures need to be verified before recomputing the aggregate which results in a small overhead (i.e., the initial aggregation and verification) with respect to the other approach. Note, though, that an attacker may execute

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

Just Accepted

Name	Rountrip (ms)	Bandwidth (Mb/s)	Used in
Global	200	25	[19, 22, 23]
Regional	100	100	[15]
National	10	1,000	[15]
Heterogeneous	1–270	66–10,004	[31]

Table 3. Scenarios used in the evaluation.

such an attack at most once as this is an obvious sign of a faulty process. In future rounds the parent processes will discard the inputs from the malicious child.

Code Availability: Our implementation is publicly available in the GitHub.[2] Overall, the implementation of the functionalities described above required the addition/adaptation of ≈ 1300 lines of code to the HotStuff codebase.

8 Evaluation

In this section we evaluate Kauri across several scenarios.

8.1 Experimental Setup

All experiments were performed on the Grid'5000 testbed [7]. We used 40 physical machines, each with two Intel Xeon E5-2620 v4 8-core CPUs and 64 GB RAM. To build large networks we need to deploy multiple processes per machine. The need to co-locate multiple nodes in a single machine is a limitation of our testbed, however, in a real-world deployment, blockchain nodes also need to perform multiple tasks, such as respond to clients, participate in the mempool protocol, validate blocks, verify client transaction signatures, and execute transactions. 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, summarized in Table 3, 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*, *national*, and *heterogeneous*. The *global* deployment models a globally distributed blockchain as used in other works [19, 22, 23] with 200ms roundtrip time (RTT) and 25Mb/s bandwidth. The two other scenarios model reported industry use cases [15] 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 [31]. We model the network characteristics of each scenario using NetEm [33] and Kollaps [21].

We compare Kauri with HotStuff-bls and HotStuff-secp the two HotStuff variants discussed in §7. The idea behind using HotStuff-bls is to highlight that using a more advanced encryption scheme is not sufficient to ensure good performance as performance is limited by the star topology. Conversely, we also executed a subset of the experiments with Kauri without pipelining to assess the performance of just using trees and the BLS encryption scheme. This configuration approximates the design of Motor [23] and henceforth we refer to it at Motor*. In fact, Motor uses a tree of depth two without pipelining and the BLS encryption scheme as in Kauri. In the absence of failures, Kauri without pipelining performs very similar to Motor. We used this, as to the best of our knowledge the original prototype of Motor is not publicly available. Furthermore, while Motor only has 2 phases of consensus, our implementation of Motor uses the HotStuff pipelining to compensate for the extra phases.

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

Just Accepted

N	100	200	300	400	500	600	700	800
m	10	14	17	20	22	24	26	28
f_r	17	34	50	68	83	99	116	134

Table 4. Root fanout m used for the different system sizes N.

For most experiments, we use system sizes with $N = 100, 200, ..., 800$ 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 root fanout m for given system sizes N as outlined in Table 4. In addition to the fanout m, we also display the number of failures f_r under which Kauri is guaranteed to find a robust tree for the given configurations in our experiments.

8.2 Configuring Kauri

We now describe the values used to configure Kauri. Table 5 shows the values of the different parameters required to compute the Kauri pipelining stretch, following the rationale introduced in §5.3. 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*. These values are used to compute both the target pipelining stretch and the expected maximum speedup for each configuration. Note that, Table 5 does not include the parameter values 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.

8.3 Aggregate & Verify

We start by evaluating the impact and cost of the aggregate-then-verify and verify-then-aggregate strategies, as discussed in §7. To this end, we executed an experiment with 100 processes in the global, regional, and national scenarios.

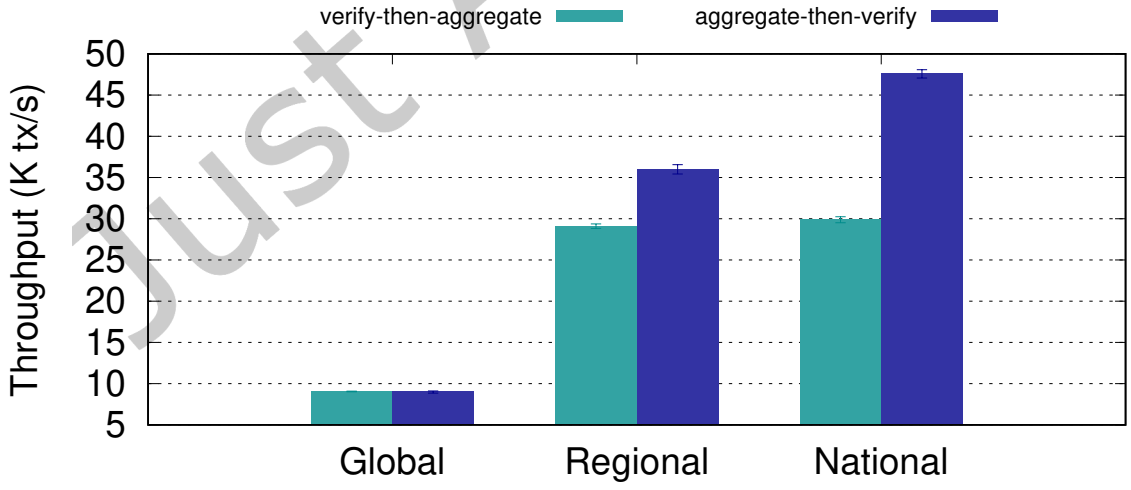


Fig. 7. Throughput for the signature verification and aggregation strategies in a system with $N = 100$.

Just Accepted

Scenario	N	Depth d	Root's fanout m	Processing time	Sending time	Remaining time	Base pipelining	Pipelining stretch	Pipelining total depth	Expected speedup
HotStuff-secp										
National	100	1	99	4	29	14	4	1	4	-
National	200	1	199	7	65	17	4	1	4	-
National	400	1	399	15	156	25	4	1	4	-
Regional	100	1	99	4	288	104	4	1	4	-
Regional	200	1	199	7	648	107	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	100	2	10	3.6	2.5	24	4	8	32	≈ 8x
National	200	2	14	5.3	3.6	25	4	6	24	≈ 12x
National	400	2	20	6.9	5.1	27	4	5	20	≈ 22x
Regional	100	2	10	3.6	25.7	203	4	9	36	≈ 11x
Regional	200	2	20	5.3	36.1	205	4	7	28	≈ 18x
Regional	400	2	14	6.9	51.6	206	4	5	20	≈ 30x
Global	100	2	10	3.6	103.0	403	4	5	20	≈ 11x
Global	200	2	14	5.3	144.3	405	4	4	16	≈ 18x
Global	400	2	20	6.9	206.3	406	4	3	12	≈ 30x

Table 5. Pipelining stretch and estimated speedup vs HotStuff-secp for a block size of $250Kb$ and different N.

The observed throughput considered scenarios is shown in Figure 7. The throughput between each strategy in the global scenario is very similar, but as we move to the regional and national scenario the advantages of the aggregate-then-verify become clear. In fact, in the national scenario the aggregate-then-verify strategy achieves ≈ 35% higher throughput than the verify-then-aggregate. This is because the bottleneck in the global scenario is the available bandwidth and hence the use of a less efficient signature aggregation strategy has less impact on the overall performance. However, in the regional and national scenarios the bottleneck shifts from the available bandwidth to the CPU and hence the effects of the more efficient aggregate-then-verify strategy become more noticeable. Considering these results, we use the aggregate-then-verify in the rest of the experiments. We will study in more detail the performance characteristics, and bottlenecks, of Kauri and the other systems in the following sections.

Just Accepted

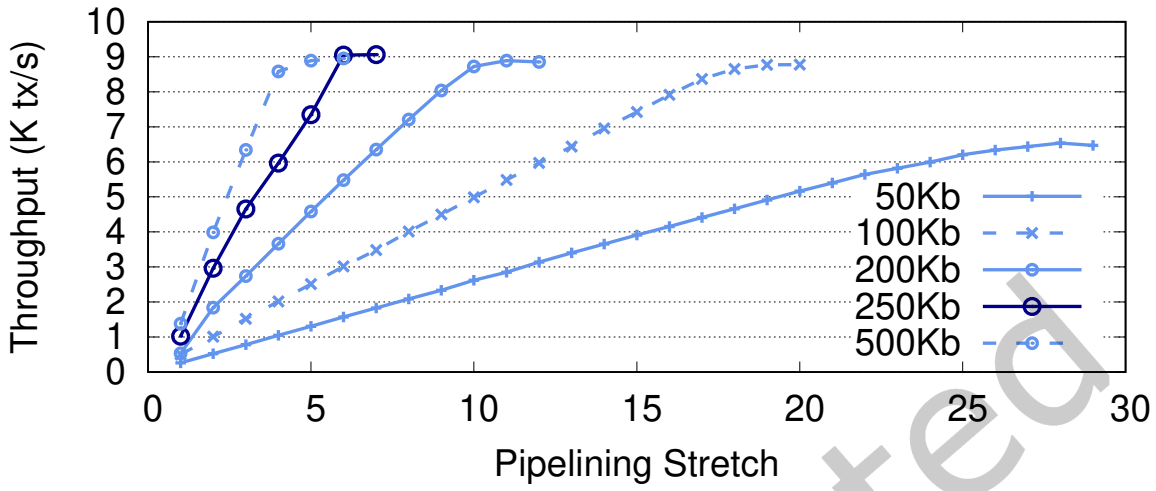


Fig. 8. Effect of pipelining stretch on Kauri's throughput for $N = 100$ and different block sizes.

8.4 Effect of Pipelining Stretch on Throughput

In the next set of experiments we analyze the effects of the pipelining stretch on Kauri's throughput. Figure 8 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 5 (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 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.

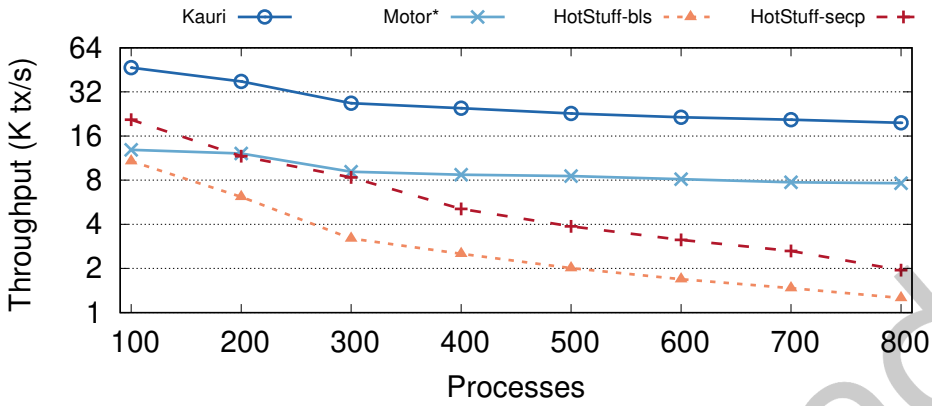
8.5 Throughput Across Different Scenarios

We now compare the throughput of Kauri and competing systems in the national, regional and global scenarios. The results are depicted in Figure 9.

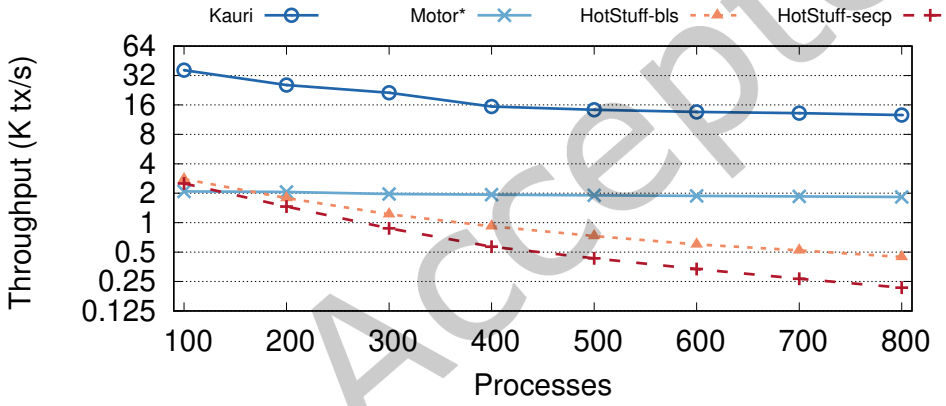
The first observation is that, as expected, HotStuff is extremely sensitive to the available bandwidth and to the total number of nodes in the system. The larger the number of nodes, the longer it takes for the leader to finish a given round. Also, for a fixed value of N, the *sending time* increases 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. In addition to that, since the use of BLS signatures reduces bandwidth usage, HotStuff-bls performs better than HotStuff-secp in most scenarios.

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 Motor* 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

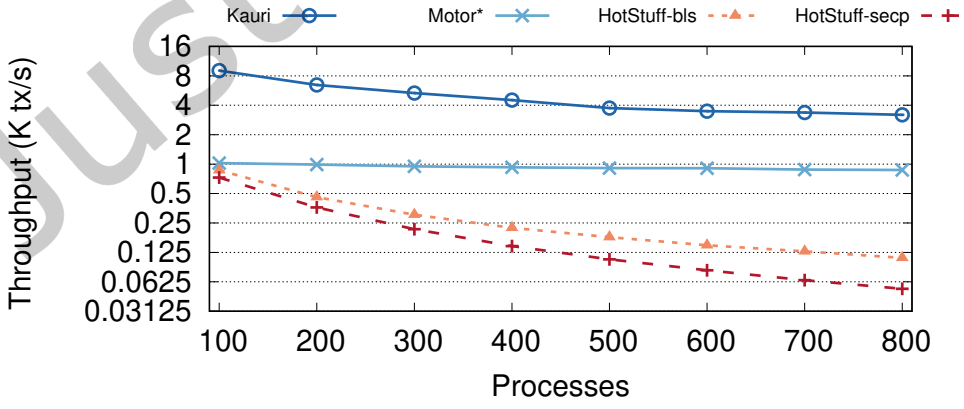
Just Accepted



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



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



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

Fig. 9. Throughput for different scenarios with a varying number of processes.

Just Accepted

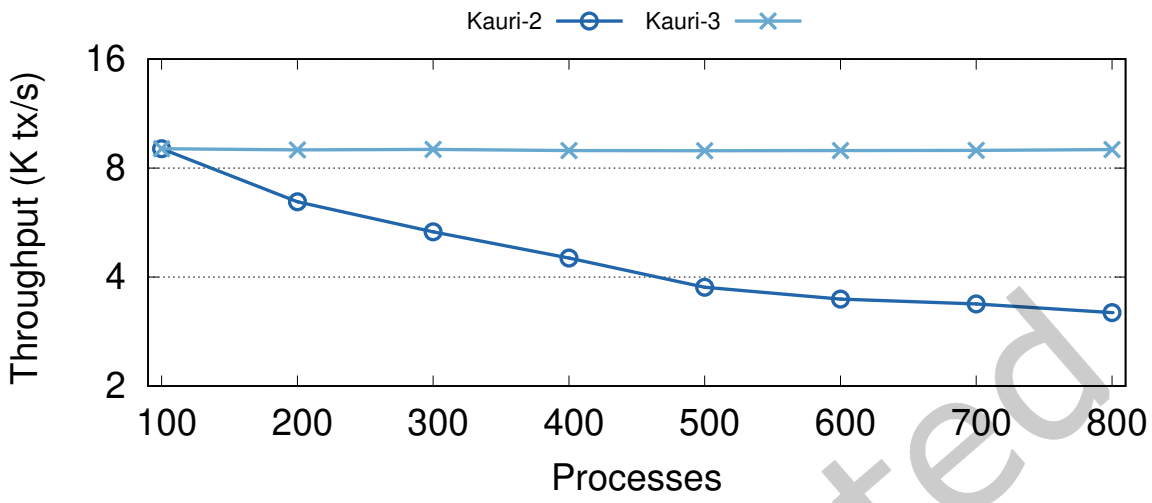


Fig. 10. Kauri throughput with increasing system sizes for different tree depths.

the network bandwidth decreases, even without our pipelining, the use of a tree already pays off. In fact, in the regional scenario, Motor* offers better throughput than both variants of 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 Motor* 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 5 we expected a speedup over HotStuff-secp of approximately $\approx 30x$ for a 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 (Motor*) results in better performance than stars in certain scenarios, only the combination of trees and pipelining permits to achieve a substantial performance increase. As a matter of fact, in a system with 800 processes, Kauri outperforms HotStuff by up to $58x$.

Finally, note that due to the use of a tree with a fixed depth of $d = 2$, Kauri's throughput drops with the number of processes. In the next experiment, we show how Kauri can maintain high throughput independently of the number of processes by maintaining a constant fanout $m = 10$ and adjusting the tree depth accordingly. The results of these experiments are shown in Figure 10 for the global scenario and, similarly to before, from 100 up to 800 processes. In this experiment, the tree with $N = 100$ processes has a depth $d = 2$ and the trees with 200 to 800 processes have a depth of $d = 3$ (note that with a fanout $m = 10$, a tree of depth $d = 2$ can have at most 111 processes, while a tree of depth $d = 3$ can have up to 1110 processes).

As expected, Kauri's throughput with a tree of depth $d = 2$ (Kauri-2 in the figure) decreases for larger system sizes (as we need to increase the fanout to accommodate more nodes, and hence the load on the internal nodes), but by increasing the tree depth to $d = 3$ (Kauri-3), the throughput remains stable as the system size increases. In general, for a fixed fanout m and tree depth d we expect Kauri's throughput to remain stable as we increase the number of processes up to the maximum number of processes supported by that fanout and depth configuration. In the following sections, we evaluate the effects of different tree fanouts and depths on latency.

Just Accepted

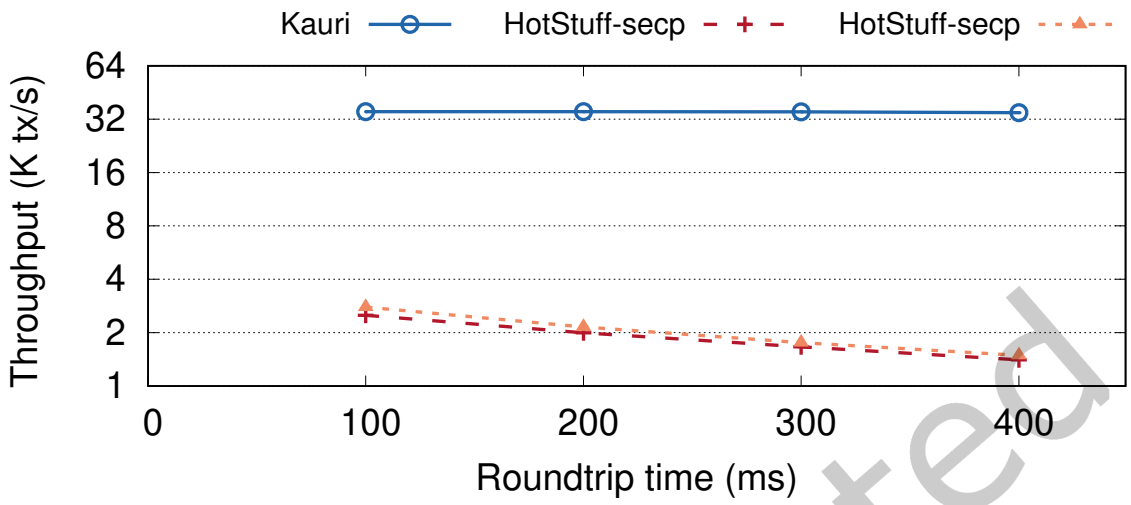


Fig. 11. Impact of RTT in system throughput. (N=100 with $100Mb/s$ bandwidth)

In most of the remaining experiments we consider practical system sizes of 100 processes to simplify the deployment and experiment times. Note that this favors HotStuff as the star topology suffers strongly in scenarios with large numbers of processes due to the bandwidth and computing bottleneck.

8.6 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 11 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 follow a similar trend.

8.7 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 would 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 12 shows the results for $N = 100$. As expected, the available bandwidth has a much larger impact on both variants of HotStuff than in Kauri. In fact, for bandwidths smaller than $100Mb/s$ Kauri offers

Just Accepted

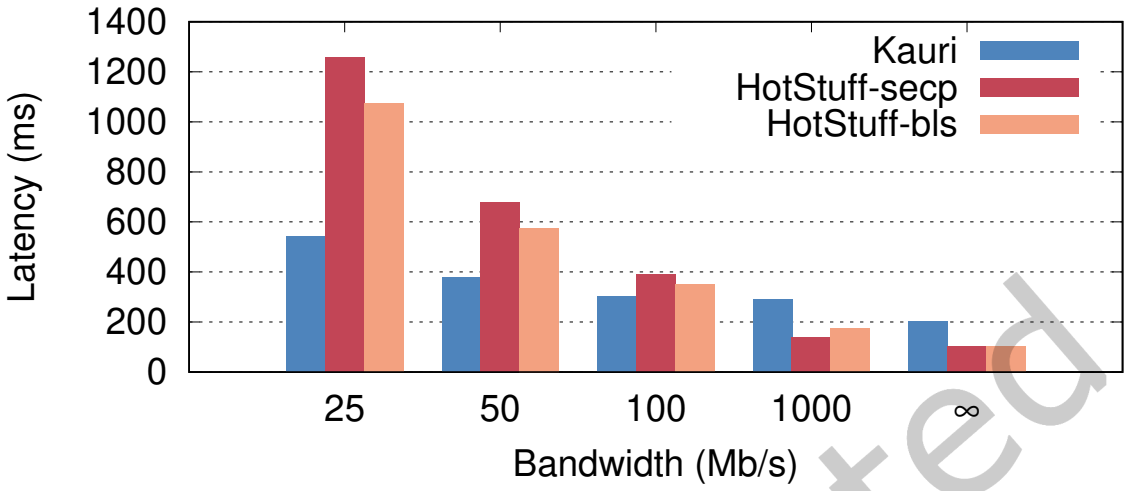


Fig. 12. Impact of bandwidth on latency (N=100, RTT=100ms).

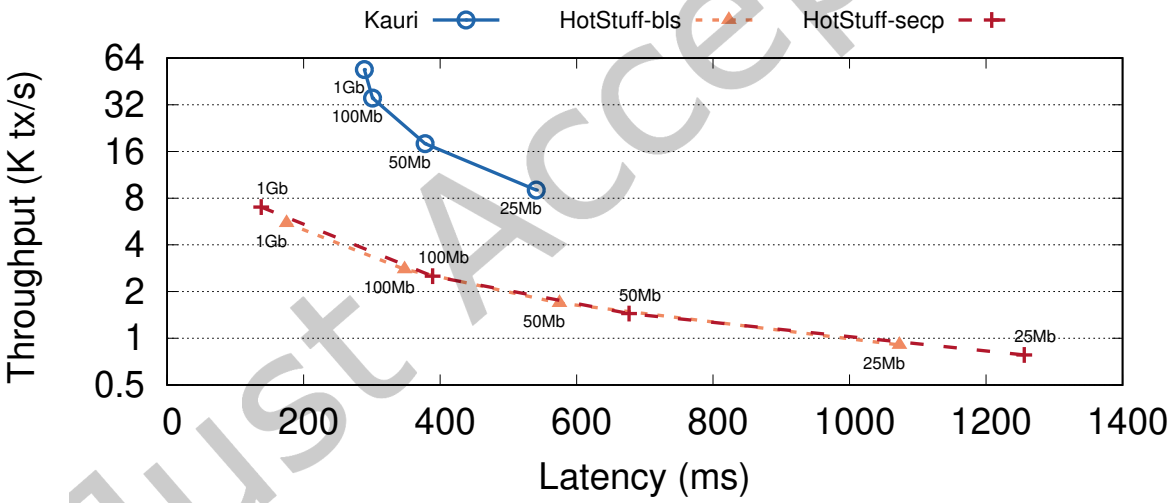


Fig. 13. Throughput/Latency tradeoffs with varying network bandwidth

Just Accepted

better latency than HotStuff, and only at high bandwidths HotStuff starts to have substantially better latency. The figure also shows analytical values for an idealized scenario of infinite (∞) bandwidth, where HotStuff's latency would be at best half of Kauri's (as we use a tree of depth $d = 2$). As in §8.6, results for the other system sizes follow a similar trend.

Figure 13 presents the results for the same experiment in a latency-throughput representation. As hinted from the previous experiments, the latency of HotStuff-secp and HotStuff-bls varies substantially with the available bandwidth whereas Kauri's are much less affected. In the scenarios where the latency of Hotstuff variants is better than Kauri's, the tradeoff is clear: for a modest latency penalty (e.g.: x2 for the 1000Mb/s scenario), we

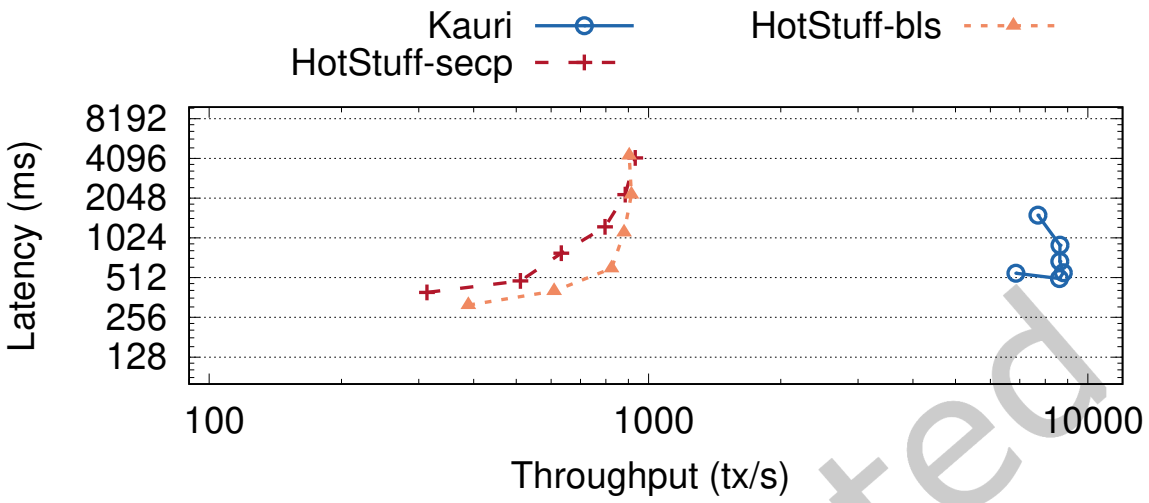


Fig. 14. Througput-Latency for varying block sizes (N=100, RTT=100ms, block size from 32Kb to 1Mb).

can achieve a substantial throughput gain (e.g.: x5 for the 1000Mb/s scenario). Interestingly, in scenarios where bandwidth is limited, Kauri is able to achieve not only better throughput but also better latency.

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.

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.

8.8 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. We use 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.

The results are depicted in Figure 14. Similarly to the previous experiments, the throughput of Kauri is substantially higher than that of either variant of HotStuff in all scenarios. As expected, as the block size increases, the latency of all systems increases due to an increase in the *sending time*. This increase is however much faster in both variants of HotStuff, and for block 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 scenarios, which also confirms the previous experiments when varying the available bandwidth (Figure 12). 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 from the CPU to the network).

Just Accepted

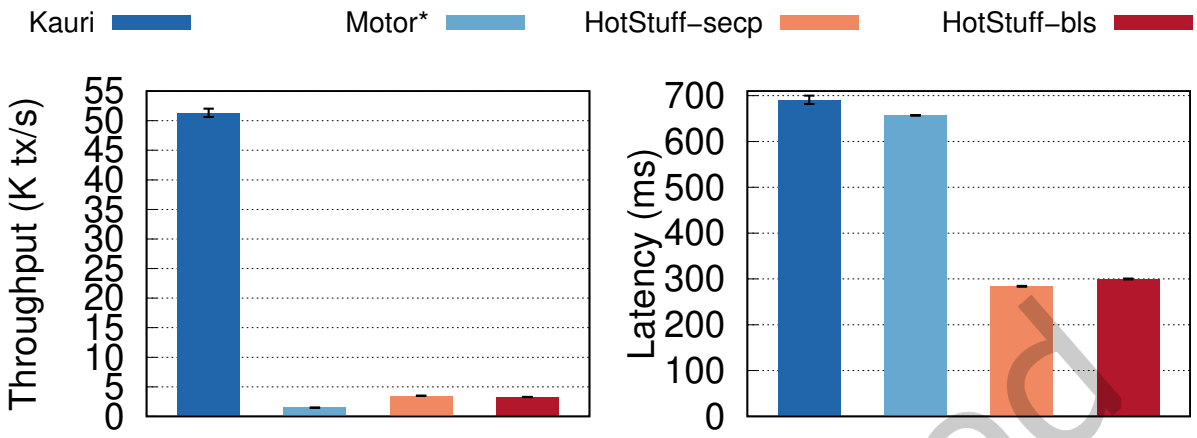


Fig. 15. Throughput in the heterogeneous scenario ([31]) with $N = 60$.

8.9 Heterogeneous Networks

Up to this point, we have studied the behavior of Kauri and the other systems in a homogeneous network where the latency and bandwidth between any two given processes are exactly the same. Next, we evaluate how these systems behave in an heterogeneous setup. As we lack the resources to execute the experiment in an actual globally distributed network, we have used the real-world scenario that was measured experimentally and used in the evaluation of ResilientDB [31].

Standard Distribution: In detail, the authors measured the latency and bandwidth between a set of Google data centers in different geographic locations, namely Oregon, Iowa, Montreal, Belgium, Taiwan, and Sydney. The latency within a datacenter is roughly 0.25ms, and the latency between datacenters ranges between 38ms (Iowa to Oregon) and 270ms (Belgium to Sydney). Similarly, the bandwidth within a datacenter is roughly 10Gb/s and varies from 66Mb/s (Belgium to Sydney) to 670Mb/s (Iowa to Oregon). We deployed 10 processes in each datacenter, resulting in a total of 60 processes hence matching one of the deployment scenarios in the ResilientDB paper [31].

As ResilientDB is deployed statically (i.e., most of the communication happens within a datacenter), we have attempted to optimize the deployment of HotStuff and Kauri accordingly. As such, the leader (root) process is always located in Oregon (best bandwidth and latency to the remaining datacenters), and we have distributed processes so that they are approximately close to their parent.

The results of this experiment are depicted in Figure 15. Similar to previous experiments, Kauri outperforms any other system in terms of throughput and, also analogous to the other deployments, this primarily stems from the use of pipelining, which allows Kauri to compensate the large round trip latencies. However, as expected, Kauri is penalized in terms of latency, as HotStuff's throughput is impaired mainly on high latency scenarios, it neither bottlenecks on bandwidth nor on the processing load. Nonetheless, Kauri displays an almost ten times higher throughput compared to only twice the latency cost.

The most interesting takeaway is that Motor* (i.e., Kauri without pipelining) exhibits the worst throughput of all approaches. This not only shows the strong negative impact that geographic distribution has on existing tree-based approaches but also highlights the importance of pipelining in order to achieve high throughput.

Just Accepted

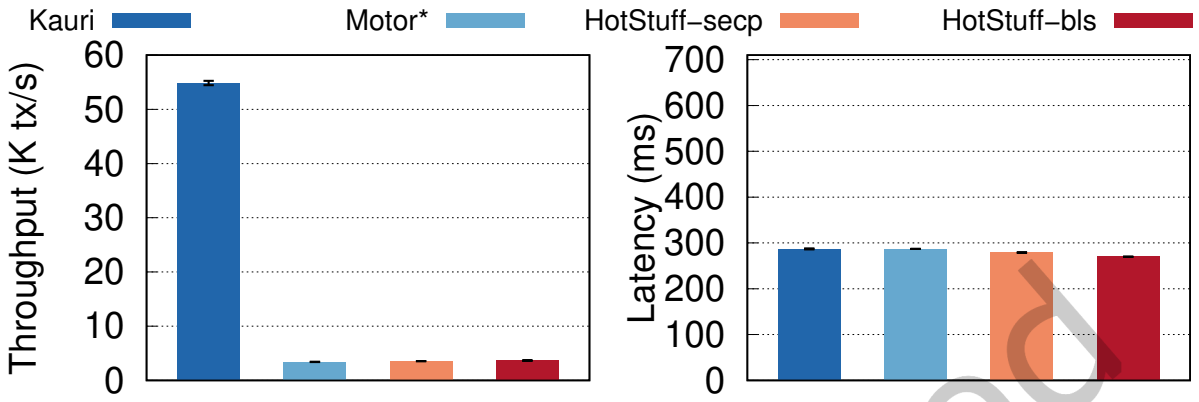


Fig. 16. Throughput in the heterogeneous scenario ([31]) with $N = 43$ and optimal node placement.

We also want to note that the values obtained for the HotStuff variants differ substantially from those reported in ResilientDB. This stems from the fact that in the evaluation in ResilientDB [31] N parallel instances of HotStuff were run, and the reported throughput is a sum of all the N parallel instances. In constrast to this, we opted to consider the best possible throughput of a single instance in our deployment. Nonetheless, the throughput we have obtained for Kauri is very similar to the throughput obtained by ResilientDB, without requiring to sacrifice the resilience of the system as ResilientDB tolerates at most $f \leq \lfloor \frac{C-1}{3} \rfloor$ failures, 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 algorithms.

While we manually assigned the leader to the Oregon datacenter and distributed the remaining internal nodes equally over the other data centers, this is still far from an optimal distribution as the internal nodes still always had to collect a rather large fraction of signatures from children located in remote datacenters. We address this in the next experiment.

Optimal Distribution: In order to achieve an optimal distribution (i.e. to minimize cross-datacenter communication), we either have to use an irregular tree or adjust the number of processes in each data center. We opted for the second option. In this experiment instead of statically assigning 10 processes to each datacenter, the Oregon datacenter (Oregon) has 8 processes (1 root node, 1 internal node, 6 leaf nodes) and the remaining datacenters have 7 processes (1 internal node, 6 leaf nodes) where each internal node is solely connected to leaf nodes within its own datacenter. This resulted in a total of 43 processes.

The result of this experiment are shown in Figure 16. We observe a slight increase in Kauri's throughput (primarily due to the reduced fanout) and a substantial reduction in latency which becomes very similar to both variants of HotStuff and Motor*. This shows that by considering the heterogeneous characteristics of the network when deploying Kauri, it is possible to achieve latency similar to HotStuff even on scenarios where the bandwidth and processing resources are not saturated.

8.10 Reconfiguration

Next, we evaluate how Kauri behaves in the presence of faults in four different scenarios: 1 faulty process, 3 faulty processes, 10 faulty processes, and f faulty processes. We have executed this experiment in the regional scenario

Just Accepted

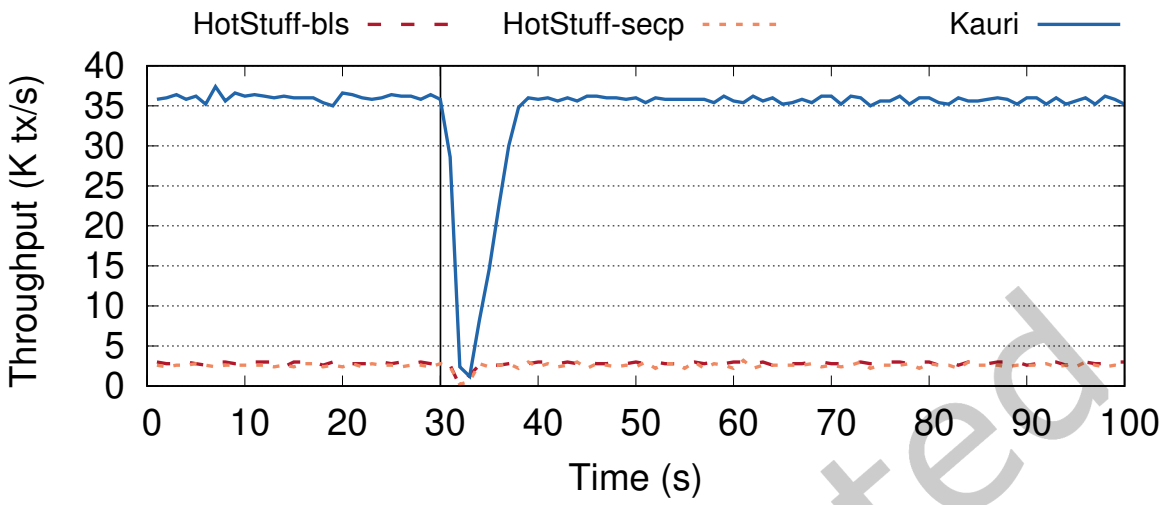


Fig. 17. Reconfiguration with one faulty leader.

(100ms RTT latency and 100Mb/s bandwidth) with 100 processes. As such, there are $f = 33$ faulty processes in the fourth scenario.

With the exception of the last scenario, we always fail consecutive leaders which forces a reconfiguration and corresponds to the worst case scenario for a given number of failures. In the first two scenarios with 1 and 3 faulty processes Kauri can recover in optimal steps. In the third scenario, we exercise the t-Bounded q-Robust Conformity given that we have a fanout of $m = 10$ and 10 faulty processes. Due to the q-Robust conformity, we are still able to recover in optimal steps in this scenario.

For the fourth scenario, we consider two variants. In the first, which we call Kauri-leaders, we fail f consecutive leaders leading Kauri to degenerate to a star. In the second variant, which we call Kauri-internal+leaders, we consecutively fail 19 internal nodes (following Theorem 4) such that a quorum is not achievable and hence forcing the degeneration to a star, and then we fail 33 consecutive leaders. Note that in this worst case scenario, some faulty nodes trigger reconfiguration twice: once as an internal node in the tree and once as a leader in the star topology. This happens because, in general, it is impossible to identify the internal node(s) responsible for triggering the reconfiguration and hence some of those nodes could be selected as leaders in the future.

As in HotStuff, reconfiguration is triggered with a timeout. We consider an initially step timeout of $1s$ ($\delta = 0.25s$) after which a process assumes the topology to be non-robust and triggers a reconfiguration. With subsequent failures this value doubles until reaching or exceeding $10s$ ($\Delta = 2.5s$) at which it is capped. In each experiment, we first have a warm-up period of $30s$ (omitted from the plots), after which we start the experiment. Following that, we inject the failure after another $30s$ (hence $60s$ after the start of the experiment) and measure the impact on the system throughput.

Results for a single and three consecutive leader failures are depicted in Figure 17 and Figure 18, respectively. As expected, both Kauri and both variants of HotSuff recover very quickly to the throughput levels before the failure. Kauri takes slightly more time than HotStuff to reach the pre-failure levels as it needs to build up the full pipeline. Nonetheless, we can see that the time this takes is independent of the number of failures. The experiment with 10 consecutive leader failures, depicted in Figure 19, follows the same trends, due to our t-Bounded q-Robust reconfiguration algorithm.

Just Accepted

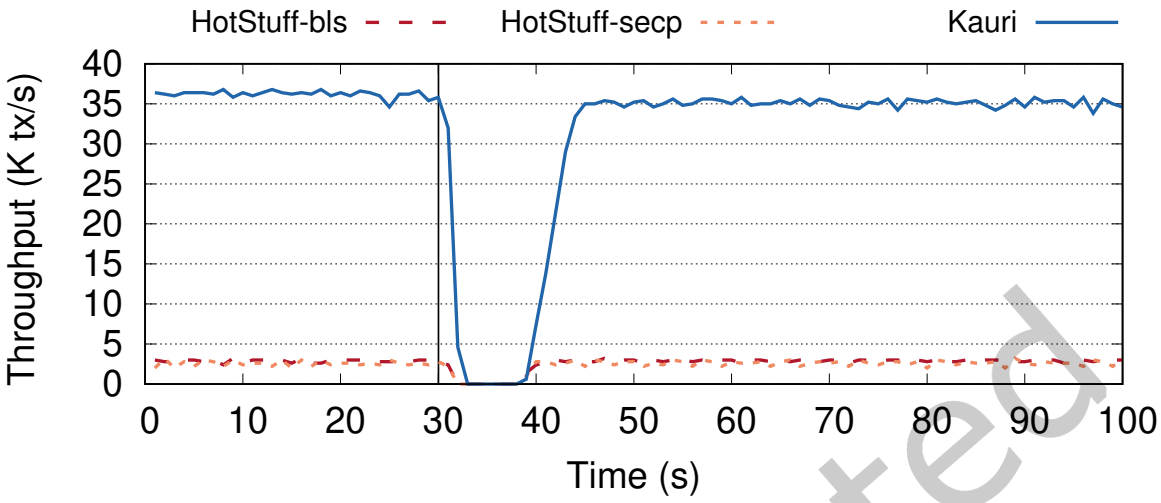


Fig. 18. Reconfiguration with three consecutive faulty leaders.

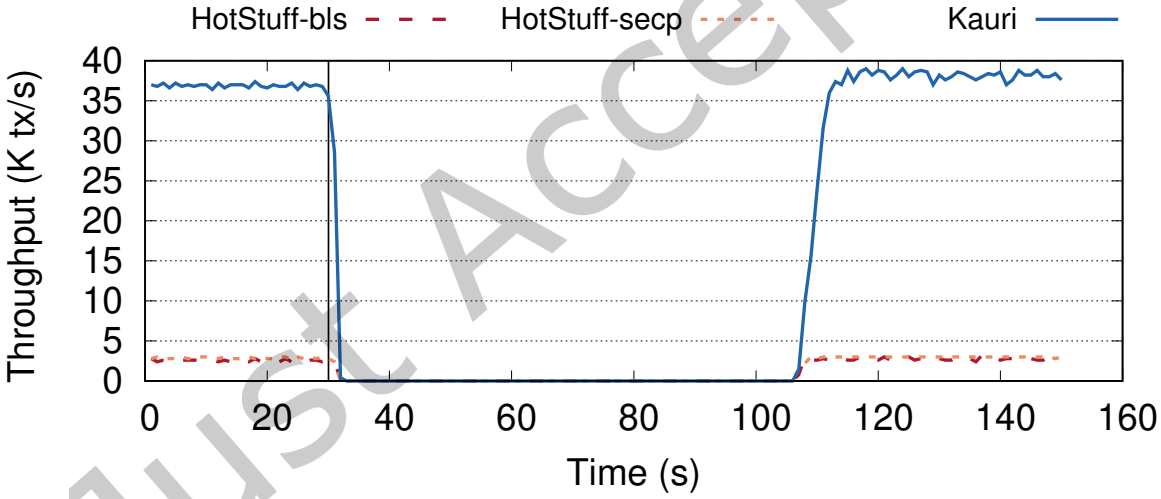


Fig. 19. Reconfiguration with ten consecutive faulty leaders.

Finally, Figure 20 illustrates the worst-case reconfiguration scenario for Kauri in the presence of $f = \frac{N-1}{3}$ faulty nodes. Although Kauri may also reach a robust tree topology in this adverse scenario (as depicted in Figure 6), in the worst case the system must revert to a star topology. In that case, the fallback takes f_a additional reconfiguration steps, introducing a latency overhead. In this experiment, while HotStuff completes recovery at the 330-second mark, Kauri requires an additional 200 seconds to finalize its reconfiguration process under these conditions.

Finally, the results for Kauri-leaders and Kauri-internal+leaders are depicted in Figure 20. In the case of Kauri-leaders, both Kauri and the HotStuff variants behave similarly and show similar recovery time. However, due to

Just Accepted

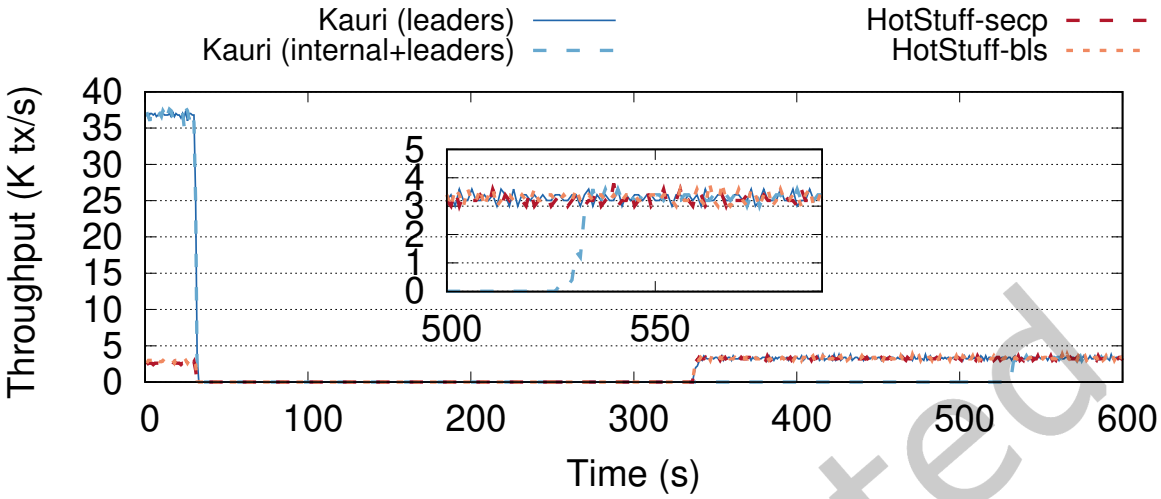


Fig. 20. Reconfiguration with $f = \frac{N-1}{3} = 33$ faulty processes.

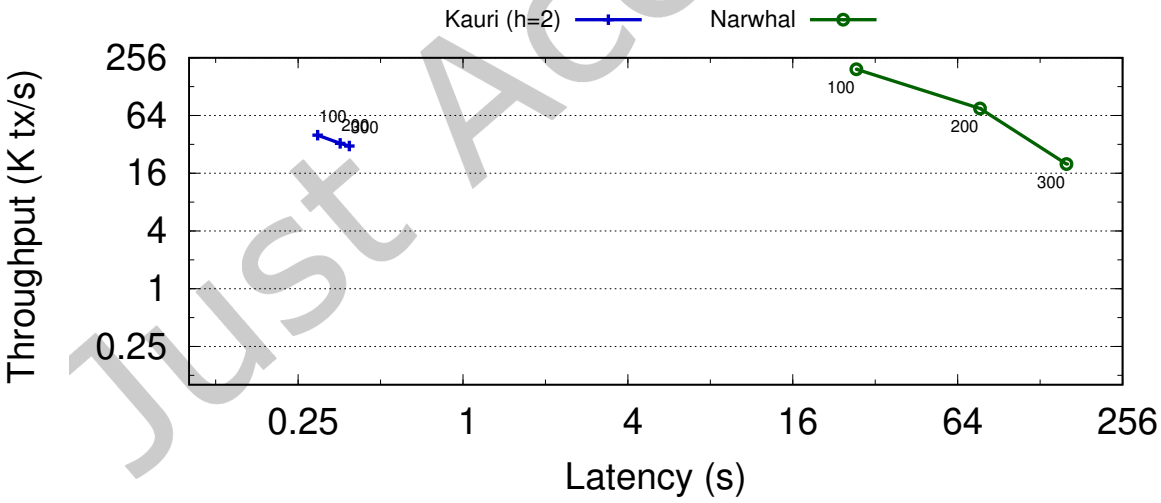


Fig. 21. Comparison between Kauri and Narwhal in the national scenario.

the many consecutive failures, Kauri has to fall back to a star topology and hence it shows performance similar to HotStuff after recovering. In the worst case of Kauri-internal+leaders, Kauri needs an an extra reconfiguration time of f_a rounds but as expected, after recovery it achieves the same performance as both variants of HotStuff.

Just Accepted

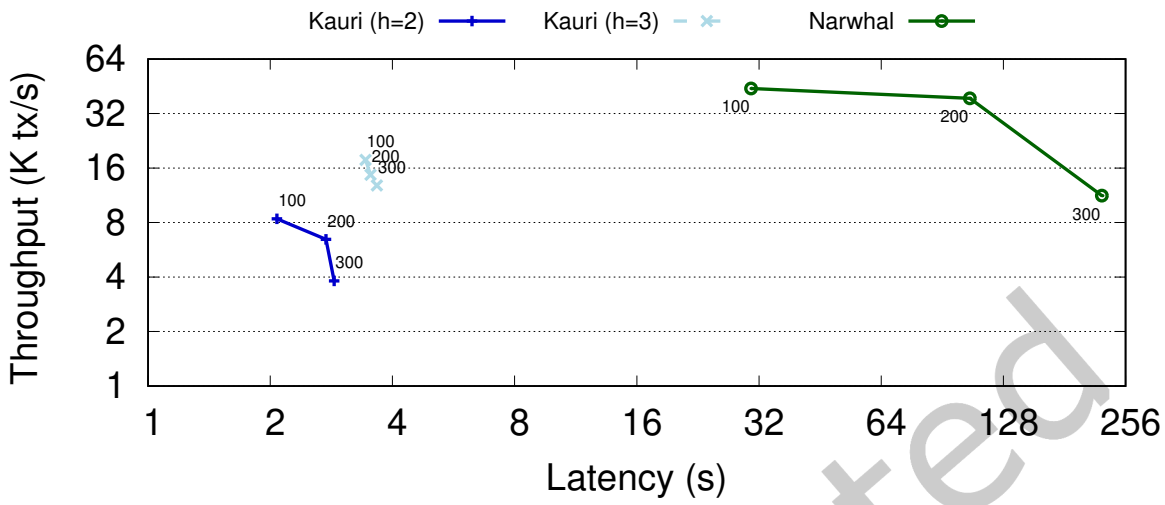


Fig. 22. Comparison between Kauri and Narwhal in the global scenario.

8.11 Comparison with DAG based Consensus

Lastly, we evaluated how Kauri compares to DAG-based approaches in terms of throughput and latency. For this evaluation, we have used the official Narwhal implementation[3].

We started by deploying both systems in the *national* scenarios, the results are presented in Figure 21. Narwhal initially achieves a significant throughput advantage, sustaining almost 5× the throughput of Kauri. However, this comes at a large latency penalty (approximately 90×). Moreover, as the number of nodes increases, Narwhal's throughput gains diminish, and Kauri surpasses Narwhal in throughput when the system size is larger than 300 processes. At the same time, the latency gap between the two systems also grows substantially.

To better understand the tradeoffs offered by each system, we conducted an additional experiment in the *global* scenario which is more constrained by bandwidth than computational resources. More specifically, we deployed Narwhal and two variants of Kauri- the configuration used in the previous experiment with a tree of depth 2 (Kauri-2) and another with a deeper tree and smaller fanout (Kauri-3). The results are depicted in Figure 22. While Narwhal initially maintains a similar throughput advantage over both Kauri configurations (with a 14× higher latency), this advantage erodes as the number of nodes increases. Eventually, Kauri with a tree height of three surpasses Narwhal's throughput at 300 nodes.

This behavior is not surprising, given that Narwhal's design uniformly distributes bandwidth load across all nodes. While this approach mitigates bandwidth bottlenecks, it incurs significant latency overhead and does not alleviate the computational load on each node.

8.12 Evaluation Summary

Our evaluation demonstrates that Kauri, by leveraging a tree structure, effectively balances the bandwidth load from block propagation and the computational load from signature verification. This enables Kauri to scale seamlessly to up to 800 nodes in our experiments, achieving a throughput advantage of up to 58× compared to the best-performing HotStuff variant. While tree structures inherently introduce additional network hops,

[3]https://github.com/facebookresearch/narwhal

Just Accepted

resulting in a moderate latency tradeoff, we show that optimal node placement can significantly mitigate this overhead. We leave the design of such an optimization mechanism to future work.

The primary tradeoff of Kauri lies in its worst-case behavior: in the presence of a large number of failures, the system may require additional reconfiguration steps and, in extreme cases, fall back to a star topology. However, in real-world production systems like Cosmos and Ethereum, participation rates are typically very high since nodes have an incentive to participate in the system and follow the protocol [5]. As a result, we expect Kauri to operate predominantly in its efficient tree structure, with fallbacks to a star topology being exceedingly rare.

Finally, when compared to modern systems like Narwhal, Kauri does not distribute bandwidth as evenly as Narwhal which achieves near-uniform bandwidth distribution across all participating nodes at the cost of significantly higher latency. However, systems like Narwhal still face a computational bottleneck, where every node must verify all signatures, severely limiting their scalability. In contrast, Kauri offers superior throughput and lower latency at larger system scales, making it a compelling choice for high-performance high-scale deployments. In the presence of failures, Narwhal-HotStuff would perform similarly to HotStuff, requiring the same number of reconfiguration steps. In contrast, Narwhal-Tusk, which leverages a leaderless asynchronous consensus protocol and only guarantees probabilistic safety would experience a performance degradation proportional to the failure rate, but does not require reconfigurations, as demonstrated in the Narwhal paper.

9 Discussion and Limitations

In this section, we discuss some possible extensions to Kauri that, while orthogonal to the contributions of this work, could broaden the deployment opportunities of Kauri.

9.1 Dynamic Pipelining

Throughout the evaluation, we assumed static network conditions. However, in practice, latency, bandwidth, and processing time of each block may change over time. Therefore, it may be useful to augment Kauri such that it can dynamically adjust the pipelining depth based on network and workload conditions.

We experimentally observed that over-pipelining increases the round-trip latency observed at the leader without affecting the overall system throughput. Based on this observation, a possible simple strategy to dynamically adapt the pipelining depth could be as follows: Periodically, the leader increases the pipelining depth and observes the resulting round-trip latency. If the latency remains unaffected, the leader adopts the new depth, otherwise, it reverts to the previous configuration. Similarly, if the leader observes an increase in latency, it can reduce the pipelining depth and assess whether this adaption was able to reduce the latency without degrading throughput.

9.2 Heterogeneous Network

Second, as discussed in the evaluation, our tree construction algorithm does not take heterogeneous network conditions into account. While, as we have shown, this does not affect the throughput of the system, in many deployments the resulting tree might exhibit a higher latency than possible due to a suboptimal deployment. However, building a robust tree whose internal nodes are no longer picked at random, requires careful consideration not only in terms of performance but also to mitigate the potential negative impact of Byzantine nodes that might attempt to force some configurations in favor of others.

9.3 Random Tree Construction

In Kauri, similar to HotStuff, the sequence of reconfigurations is deterministic. This allows a powerful adversary to make sure that the system always takes the worst case number of reconfiguration steps to reconfigure. As such, an interesting extension could introduce some randomness in the tree construction, for example, with the help of verifiable random functions as in [26].

Just Accepted

9.4 2-Phase HotStuff

HotStuff-2 [28] improves upon HotStuff by reducing consensus latency from three phases to two phases. The optimization leans on the observation that a leader can propose a new block as soon as it receives a double quorum certificate for the preceding block. Kauri uses a stable leader approach and, therefore, can also benefit from this optimization in steady state. HotStuff-2 also introduces a protocol change where processes send the votes to the next leader to ensure that this condition is always met after GST, even when there is one or more view-changes, as long as the (new) leader is correct. In Kauri it is harder to ensure that this condition is met during a view-change because collecting a quorum certificate after GST is not enough to have a correct leader; the aggregation tree needs also to be q−*Robust*. Therefore, we leave incorporating this optimization in Kauri as future work.

9.5 Narwhal-Kauri Hybrid

As shown in the evaluation, when signature verification becomes the bottleneck (i.e., when consensus is CPU-bound), Kauri can outperform Narwhal in terms of throughput. However, the Narwhal dissemination mechanism does a better job in distributing the communication load when propagating the block payload (in Kauri, leaf nodes do not contribute to message dissemination). To get the best out of both worlds, we envision a Narwhal-Kauri hybrid that combines DAG-based consensus for block dissemination with Kauri for signature aggregation. Similar to the deployment of Narwhal-HotStuff, we expect such a design to further increase the throughput of Kauri, at the cost of higher latency. We leave the development and evaluation of this variant for future work.

10 Conclusions

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 moderate, while still ensuring that a robust configuration is found in a linear number of steps for any number of faults $f \leq \frac{N-1}{3}$. In contrast to solutions based on committees, Kauri does not compromise the resilience or the finality of consensus. Kauri's throughput substantially outperforms HotStuff's in all considered scenarios, reaching up to 58x with only a modest increase in latency. In bandwidth-constrained scenarios Kauri outperforms HotStuff in both throughput and latency. Finally, Kauri can also outperform modern DAG-based systems such as Narwhal [14] for a larger number of participants.

Acknowledgements: We thank the ACM TOCS Editors and the revieers for their help in improving 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. PTDC/EEI-COM/29271/2017, 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, grant agreement number 952226, and project GLOG, with reference LISBOA2030-FEDER-00771200, and project BIG (Enhancing the research and innovation potential of Tecnico through blockchain technologies and design Innovation for social Good), and project ScalableCosmosConsensus, and the Austrian Science Fund (FWF) SFB project SpyCoDe F8502 and the Vienna Science and Technology Fund (WWTF) project SCALE2 CT22-045.

References

[1] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. 2010. Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. *IEEE Transactions on Dependable and Secure Computing* 7, 1 (2010), 80–93.

Just Accepted

[2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 30, 15 pages. doi:10.1145/3190508.3190538

[3] B Artur, Yu M Ermol'ev, and Yu M Kaniovskii. 1983. A Generalized Urn Problem and its Applications. *Cybernetics* 19, 1 (1983), 61–71.

[4] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.* 32, 4, Article 12 (Jan. 2015), 45 pages. doi:10.1145/2658994

[5] Zeta Avarikioti, Eleftherios Kokoris Kogias, Ray Neiheiser, and Christos Stefo. 2025. CoBRA: A Universal Strategyproof Confirmation Protocol for Quorum-based Proof-of-Stake Blockchains. arXiv:2503.16783 [cs.CR] https://arxiv.org/abs/2503.16783

[6] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. 2024. Mysticeti: Reaching the Limits of Latency with Uncertified DAGs. arXiv:2310.14821 [cs.DC] https://arxiv.org/abs/2310.14821

[7] D. Balouek, A. Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec. 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, Berlin, Heidelberg, 3–20.

[8] J. Bernstein, T. Lange, et al. 2013. SafeCurves: Choosing Safe Curves for Elliptic-Curve Cryptography. http://safecurves.cr.yp.to. Accessed on 18.04.2022.

[9] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, Atlanta, GA, USA, 355–362.

[10] Dan Boneh, Ben Lynn, and Hovav Shacham. 2004. Short signatures from the Weil pairing. *Journal of cryptology* 17, 4 (2004), 297–319.

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

[12] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, USA, 173–186.

[13] Bram Cohen and Krzysztof Pietrzak. 2020. The chia network blockchain. https://www.chia.net/assets/ChiaGreenPaper.pdf Accessed on 18.04.2022.

[14] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 34–50. doi:10.1145/3492321.3519594

[15] M. del Castillo. 2020. Forbes Blockchain 50. https://www.forbes.com/sites/michaeldelcastillo/2020/02/19/blockchain-50/?sh=77ed26207553 Accessed on 18.04.2022.

[16] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (4 1988), 288–323.

[17] Michael Eischer and Tobias Distler. 2018. Latency-Aware Leader Selection for Geo-Replicated Byzantine Fault-Tolerant Systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, Luxembourg, Luxembourg, 140–145.

[18] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *Journal ACM* 32, 2 (4 1985), 374–382.

[19] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 51–68.

[20] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, Portland, OR, USA, 568–580.

[21] Paulo Gouveia, João Neves, Carlos Segarra, Luca Liechti, Shady Issa, Valerio Schiavoni, and Miguel Matos. 2020. Kollaps: Decentralized and Dynamic Topology Emulation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. ACM, New York, NY, USA, Article 23, 16 pages.

[22] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 279–296.

[23] Eleftherios Kokoris-Kogias. 2019. Robust and Scalable Consensus for Sharded Distributed Ledgers. *IACR Cryptology ePrint Archive* 2019 (2019), 676.

[24] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA,

Just Accepted

583–598.
[25] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982), 382–401.
[26] Peilun Li, Guosai Wang, Xiaoqi Chen, Fan Long, and Wei Xu. 2020. Gosig: A Scalable and High-Performance Byzantine Consensus for Consortium Blockchains. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. ACM, New York, NY, USA, 223–237.
[27] 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.
[28] Dahlia Malkhi and Kartik Nayak. 2023. Extended Abstract: HotStuff-2: Optimal Two-Phase Responsive BFT. Cryptology ePrint Archive, Paper 2023/397. https://eprint.iacr.org/2023/397
[29] Ray Neiheiser, Daniel Presser, Luciana Rech, Manuel Bravo, Luís Rodrigues, and Miguel Correia. 2018. Fireplug: Flexible and robust N-version geo-replication of graph databases. In *2018 International Conference on Information Networking (ICOIN)*. IEEE, Chiang Mai, Thailand, 110–115.
[30] W. L. Nicholson. 1956. On the Normal Approximation to the Hypergeometric Distribution. *The Annals of Mathematical Statistics* 27, 2 (1956), 471–483. http://www.jstor.org/stable/2237005
[31] Sajjad Rahnama, Suyash Gupta, Thamir M. Qadah, Jelle Hellings, and Mohammad Sadoghi. 2020. Scalable, resilient, and configurable permissioned blockchain fabric. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 2893–2896. doi:10.14778/3415478.3415502
[32] Thomas Ristenpart and Scott Yilek. 2007. The Power of Proofs-of-Possession: Securing Multiparty Signatures against Rogue-Key Attacks. In *Advances in Cryptology - EUROCRYPT 2007*, Moni Naor (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 228–245.
[33] HEMMINGER S. 2005. Network Emulation with NetEm. https://cir.nii.ac.jp/crid/1572543024894323456. Accessed on 18.04.2022.
[34] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. 2022. Mir-bft: Scalable and robust BFT for decentralized networks. *Journal of Systems Research* 2, 1 (2022).
[35] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. 2022. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 17–33. doi:10.1145/3492321.3519579
[36] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. 2009. Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*. IEEE, Niagara Falls, NY, USA, 135–144.
[37] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2010. EBAWA: Efficient Byzantine Agreement for Wide-Area Networks. In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*. IEEE, San Jose, CA, USA, 10–19.
[38] Marko Vukolić. 2016. The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication. In *Open Problems in Network Security*, Jan Camenisch and Doğan Kesdoğan (Eds.). Springer, Cham, 112–125.
[39] P. Wuille. 2018. libsecp256k1. https://github. com/bitcoin/secp256k1. Accessed on 18.04.2022.
[40] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. doi:10.1145/3293611.3331591

Received 16 May 2022; revised 18 April 2025; accepted 5 September 2025

Just Accepted