Self-Adapting BFT Consensus: Leveraging Heterogeneity in Dissemination/Aggregation Trees

(extended abstract of the MSc dissertation)

Helena Teixeira
Instituto Superior Técnico, Lisboa, Portugal

Supervisors: Professor Luís Rodrigues e Professor Miguel Matos

Abstract

Permissioned blockchains are a class of blockchains where the processes that run consensus are limited and known by all participants. These blockchains can execute variants of *Byzantine consensus* that offer *finality*. An approach to support a large number of participants in this context is to use dissemination and aggregation trees to support the communication required to execute protocol rounds. Previous work using this topology is either topology-agnostic or assumes homogeneous environments. Many Byzantine consensus protocols are leader-based, and in the blockchain scenario, there are compelling reasons to rotate the leader between consecutive consensus instances, such as the distribution of the load of the leader and censorship resistance. This work addresses the challenges of implementing a rotating leader policy in BFT consensus that uses dissemination and aggregation trees, and we propose topology-aware heuristics to create dissemination and aggregation trees in heterogeneous environments. Through simulations, we evaluate the performance of our heuristics in realistic scenarios, showing that they can reduce the average time needed to collect a quorum by 70%.

Keywords: Byzantine Fault Tolerant Consensus, Dissemination and Aggregation Trees, Heterogeneity

1. Introduction

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

It was recently shown that dissemination/aggregation trees could support the data exchange required by blockchain consensus while distributing the load and avoiding bottlenecks [12]. Previous tree-based work, however, is either agnostic to network topology [3, 16] or assumes homogeneous environments [12]. In particular, the Kauri [12] system creates random dissemination trees that are oblivious to the topology and heterogeneity of the network. The tree configuration can poorly impact the system's performance if deployed in a global setting.

However, using trees in either context increases consensus rounds' latency. Therefore, pipelining techniques have been suggested to mitigate the impact of the increased latency in the system throughput. Kauri [12], like many other BFT protocols, is a leader-based protocol that combines these two techniques. Nevertheless, to maximize the benefits of pipelining, it uses a stable leader policy, i.e., the same leader is used for consecutive instances of consensus until this leader is suspected to be faulty. Furthermore, the same tree is used while a given leader is active.

Pipelining is also used in protocols such as Hotstuff[16], which relies on a star topology and starts the next consensus instance optimistically while the previous instance is still running and piggybacking messages from multiple instances on the same network packets.

Several arguments support changing the leader (and, consequently, the tree) between consecutive consensus instances. For instance, in a blockchain setting, the leader can select which transactions are included in a block and, therefore, has the

power to censor some specific transactions; the rotation of the leader task among all participants provides censorship resistance. Changing the leader also allows for a better distribution of the load on the leader during the protocol. Nevertheless, changing the leader frequently raises concerns about maintaining the pipeline started by the previous leader.

2. Related work

In this section, we will review our analysis of the state of the art, focusing on six key factors depicted in Table 1. The first two factors capture the communication pattern used to execute rounds and the strategy used to change leaders. Next, for the *Non-deterministic sequence* factor, we focus on whether systems use a deterministic sequence of leaders; systems that use a deterministic sequence of leaders may be more vulnerable to attacks, given that attackers can exploit this knowledge to their benefit. The *Leverages pipelining* factor captures if the systems use pipelining techniques to improve throughput. The following factor is regarding load balancing. We consider two definitions of load balancing in this discussion: intra-round load balancing (the load is distributed within a round) and cross-round load balancing (the load is distributed in the overall execution across multiple rounds). Regarding load balancing on nodes, Kauri [12] offers intra-round load balancing by distributing the load in a tree topology. On the other hand, various systems like PBFT [3] and Spin [15] provide cross-round load balancing. Here, the leader's load is distributed per all processes within several rounds. Therefore, in the *Load balancing on nodes* factor, we focus on the cross-round definition, whether the system can distribute the computational and bandwidth usage among different nodes across rounds. The last factor indicates whether knowledge regarding the network conditions is used to achieve better performance.

We described two systems with a clique topology: PBFT [3] and Spin [15]. PBFT still has some issues that can be improved, like the number of resources needed, the vulnerability to performance attacks [2], and the significant message complexity of the protocol. Spin tries to solve the vulnerability to performance attacks. It uses a rotating strategy with a blacklisting technique for the processes suspected to be faulty. This technique allows for a rotating approach but without a faulty process periodically becoming the leader. Other solutions like Aardvark [4] also solves problems related to attacks on performance. However, its solution to prevent faulty leaders from delaying the service is less efficient than Spinning. Aardvark also changes the leader when it suspects faulty behavior by running a view change operation. However, Spinning does not incur the cost of running a distributed algorithm with several communication steps. As a result, Spin becomes more efficient than Aardvark. In addition, it allows for load balancing among the processes across rounds.

For star topology, we described three more systems: Hotstuff [16], Prosecutor [17], and Carousel [5]. Hotstuff primary advantage is simplicity, enabling pipelining techniques and building large-scale replication services. Some disadvantages are the bottleneck on the leader and the leader election being performed by following a round-robin sequence. A faulty process will periodically be the leader. Also, this round-robin sequence is deterministic and predictable, making the system vulnerable to attacks on the leader. Prosecutor focuses on the impact of having a faulty leader periodically and uses a dynamic-penalization election technique to mitigate this problem, reducing the time the system is unavailable. However, it imposes computational costs on faulty servers, which is a different approach from others since the previous approaches on penalization allow faulty servers to be freely released after pretending to be correct servers. On the other hand, PoW-like penalization may become less efficient if faulty servers have a strong computation capability. Carousel also presents a new technique to deal with the impact of the round-robin approach. It focuses on minimizing the effect of crash-only executions by a participation tracking technique. This new leader-rotation mechanism demonstrated drastic performance improvements in throughput and latency compared with Hotstuff-based systems with a round-robin mechanism.

In the category of Random topology, we mentioned two systems: Gosig [9] and Algorand [6]. These two systems provide solutions to the vulnerability of attacks on the leader. Gosig focuses on minimizing attacks on the leader using VRF so attackers do not know the sequence. This method means the leader is only known when the consensus round starts, so the system tolerates attacks on the leader. Algorand selects consensus groups, using a VRF to prevent attacks targeting the leaders. However, this system has the drawback of using a committee approach for consensus, which limits the system's resilience.

Categorized as a tree topology, we analyzed two existing systems: Byzcoin [8] and Kauri [12]. Both systems use a tree communication pattern to distribute the computational and bandwidth load among processes intra-round. Byzcoin builds trees, using a VRF, to collect signatures when a miner creates a block. However, the throughput depends on the round latency and does not provide a quick recovery. The main advantages of Kauri are

	Topology	Leader election approach	Non-deterministic sequence	Leverages pipelining	Load balancing on nodes	Aware of network conditions
PBFT [3]	Clique	stable	✗	✗	✗	✗
Spin [15]	Clique	rotating	✗	✓	✓	✗
Prosecutor [17]	Star	stable	✗	✗	✗	✗
Hotstuff [16]	Star	rotating	✗	✓	✓	✗
Carousel [5]	Star	rotating	✗	✗	✗	✗
Gosig [9]	Random	rotating	✓	✓	✗	✗
Algorand [6]	Random	rotating	✓	✗	✓	✗
Kauri [12]	Tree	stable	✗	✓	✗	✗
Byzcoin [8]	Tree	rotating	✓	✗	✓	✗
GeoBFT [7]	Hierarchical	stable	✗	✗	✗	✓
Kauri Adaptive	**Tree**	**rotating**	✓	✓	✓	✓

Table 1: Comparison of existing algorithms.

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

Lastly, with a hierarchical topology, we analyzed GeoBFT [7] that provides a topological-aware grouping of replicas in local clusters, introducing parallelization of consensus at the local level and minimising communication between clusters to achieve better performance when the system is distributed globally.

3. Kauri Adaptive

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

3.1. Model and Assumptions.

As a model, we assume the same assumptions as Kauri, which are the standard for this type of system [12, 3, 16]. In particular, we assume a system consisting of N processes, f of which may be Byzantine subject to the constraint $f < N/3$. Byzantine processes can behave arbitrarily but do not have sufficient computational power to subvert the cryptographic primitives. Additionally, we assume the existence of a system capable of providing the distances, in terms of latency, between the different processes in the system. In practice, this can be achieved with coordinate systems such as Newton [14].

3.2. Architecture

We divided our approach into two distinct problems: i) generate all trees (described as the *Generation phase*); ii) given the set of generated trees, define their sequence (described as the *Sequence phase*).

Next, we will describe each one in more detail.

3.2.1 Generation Phase

In the *Generation phase*, our goal is to generate trees that meet the following requirements: (i) each tree generated is aware of its topology; and (ii) each generation considers the heterogeneity of the environment.

Our approach to generating a topology-aware tree starts from two observations: i) the Internet presents characteristics of a *small-world* network [1] and ii) nodes participating in permissioned blockchain systems are typically clustered in geographically dispersed data centers [7] that have internally very low latencies and between them substantially higher latencies as illustrated in Table 2. Thus, the intuition of our approach is that at the beginning of the dissemination, we spread the information over geographically distant areas and, therefore, potentially using links with higher latency, and from then on, take advantage of the locality and do the rest of the dissemination using local links that typically have lower latencies. The detailed implementation of these algorithms is specified in Section 3.3. The decision process on which heuristic to use is as follows: Can I manipulate the environment in terms of the distribution of nodes through the data centres?

Latência (ms)						
	O	I	M	B	T	S
Oregon (O)	$\leqslant 1$	38	65	136	118	161
Iowa (I)		$\leqslant 1$	33	98	153	172
Montreal (M)			$\leqslant 1$	82	186	202
Belgium (B)				$\leqslant 1$	252	272
Taiwan (T)					$\leqslant 1$	137
Sydney (S)						$\leqslant 1$

Table 2: Communication latency between different Google data centers as measured by ResilientDB [7].

1. If not, we use a Diversity-Aware group distribution to generate the groups and, for each group, create trees with a Generic tree configuration;

2. If we can, we use a Bandwidth-aware group distribution to generate the groups and create trees with a tree configuration that focuses on local dissemination for each group.

This will result in N trees generated.

3.2.2 Sequence Phase

In the *Sequence phase*, we focus on creating the sequence of the generated trees. Our goal is to create a strategy that meets the following requirements: (i) the trees have different roots, meaning that after all the generations, the extra load imposed on the root is balanced by all nodes; (ii) leverages the pipeline by guaranteeing that the leader in configuration *i* was an internal node in the configuration *i - 1*; and (iii) the sequence is not predictable for an attacker to exploit the system.

From the N-generated trees, our approach leverages solutions that use Verifiable Random Functions like Algorand [6] and uses them to choose arbitrarily between the direct children of the previous root, as exemplified in Algorithm .1.

This way, we (i) have a non-deterministic sequence; (ii) assuming the function is uniform in the long term, we guarantee fairness; and (iii) maintain the pipeline by choosing from direct children of the previous root.

Algorithm .1: VRF logic to add in Kauri

Input: T - set of trees generated for each node. r - round.
Output: Next tree $T[k]$.

```
1  if In round r − 1 then
2      for k ∈ currentTree.children do
3          vrfOutput_k,
             proof_k ← VRF(SK, seed);
4  if goToNextRound then
5      max ← MINHASH;
6      nextTreeLeader ← ∅;
7      for k ∈ currentTree.children do
8          hash ← HASH(vrfOutput_k);
9          if max ≤ hash then
10             max ← hash;
11             nextTreeLeader ← k;
12     return T[nextTreeLeader]
```

3.3. Topology-Aware Trees

In this section, we present in detail the heuristics developed to generate Topology-Aware Trees. We start with two heuristics for allocating nodes to groups: (a) the Diversity-Aware distribution and (b) the Bandwidth-Aware distribution. Finally, we show the implementation of the algorithm for building trees from a group in two scenarios: (a) a generic one, where we don't have control of the environment and want to generate the best possible tree for that environment; and (b) a heuristic using local-dissemination, where we have control of the environment and can manipulate it to a specific high-performing setup.

3.3.1 Allocating Nodes to Groups

Since Kauri takes a fixed set of groups as its starting point, the first challenge is determining how to allocate nodes to groups. To do this, we start by introducing the concept of δ-Aglomeration:

δ-Aglomeration A δ-Aglomeration is a set of nodes that are at most δ latency units apart.

Starting from the information provided by the geographic coordinate system and the δ specified by the administrator, we use a clustering [13] algorithm to divide the nodes into several δ-agglomerates. In practice, this roughly corresponds to dividing the nodes by the data centres or geographic proximity in which they are located.

Next, we present two heuristics to distribute the nodes within the groups.

Diversity-aware Distribution In this heuristic, we create t groups initially empty, which correspond to the groups used by Kauri to define the internal nodes of the tree (lines 1– 2 of the Algorithm .2). Starting from the δ-agglomerates previously constructed, we distribute the nodes of each δ-agglomerate across the B_i-groups in a rotating manner and defined in the lines 3– 5 of the Algorithm .2. Thus, by spreading the nodes of each δ-agglomerate across the different groups, we maximize the diversity of each B_i-group. This fact will be exploited in the next steps of building a tree from a group, like in the heuristics .4.

Algorithm .2: Alocation of nodes to groups

```
1 for i = 1, 2, ... (N/I) do
2     B_i ← ∅;
3 for i = 1, 2, ..., I do
4     n ← node from δ-aglomerate not
        allocated yet ;
5     B_i ← B_i ∪ {n} ;      /* Group i with I
        nodes created */
```

Bandwidth-aware Distribution This heuristic depends on having the bandwidth measures provided by the geographic coordinate system for each δ-aglomerate and the block size in the system. With this, we calculate the number of internal nodes per δ-aglomerate i as follows:

$$I_i = \frac{bandwidth_i}{blocksize} + 1$$

Resulting in the number of internal nodes in a group, I, to be the sum of the calculated factors for each δ-aglomerate i. We create t groups initially empty, which correspond, again, to the groups used by Kauri to define the internal nodes of the tree (lines 2– 2 of the Algorithm .3). From the number of internals per δ-aglomerate calculated previously, we add that amount of internal nodes from each cluster to each group (lines 3– 6 of the Algorithm .3).

Algorithm .3: Alocation of nodes to groups

```
1 for i = 1, 2, ... (N/I) do
2     B_i ← ∅;
3 for i = 1, 2, ..., I do
4     for i = 1, 2 ∈ C do
5         internals ← I_i nodes from cluster i;
6         B_i ← B_i ∪ internals;      /* Group i
            with I nodes created */
```

Thus, we can minimize the latency within each local cluster by creating groups aware of the bandwidth bottlenecks. This fact will be exploited in the next steps of building a tree from a group, like in heuristics .5.

3.3.2 Building a Tree from a Group

Given a group derived from any of the previous strategies, we present how to build a tree from it. We implemented two approaches, and both are divided into two sub-problems: (a) configuring the internal nodes and (b) allocating the leaves.

Algorithm .4: Build an informed tree

Input: A list B_i with the group of I internal nodes. M - arity of the tree. A list $remaining$ with the leaves of the tree.
Output: Tree T represented as list.

```
/* Add root to tree                          */
1  root ← rootHeuristic();
2  T ← T ∪ {raiz}
   /* Global dissemination on the first
      level of the tree                      */
3  for internal = 1, 2, ... M do
4      T ← T ∪ {node from a δ-aglomerate not
         yet chosen from group B_i}
   /* Number of levels                       */
5  L ← round(log_M(N + 1)) − 1
   /* In case there is more than one
      level of internals                     */
6  parents ← deeper nodes in T while L! = 1
     do
7      for parent ∈ parents do
8          Order remaining internal nodes of B_i
             per latency to parent;
9          T ← M nodes with less latency;
10     L ← L − 1;
   /* Allocate leafs to the deeper nodes
      */
11 parents ← deeper nodes in T
12 for parent ∈ parents do
13     Order nodes of remaining per latency to
         the parent;
14     T ← M nodes with less latency;
15 return T
```

Generic Heuristic For this heuristic, we assume that the system administrator defines the number of processes N, the *fanout* of the tree, and the δ factor detailed above.

This logic is more formally defined in Algorithm .4.

Having selected the root using a convenient heuristic, the next step is to define the configuration of the remaining nodes in the B_i group that will be the internal nodes of the tree. It should be noted that the choice of the root is essential, as described above, and the order in which the internal nodes appear in the tree is also essential to achieve good performance. Specifically, dissemination in Kauri happens, by convention, from left to right. Therefore, the leftmost nodes should be those that have a lower latency connection to the root (lines 3–4 in the Algorithm .4). This is relevant because the consensus algorithm does not need to wait for all nodes to make progress. A quorum is sufficient. Thus, by placing the nodes with the

least latency for the leftmost root, we maximize the probability of not waiting for a response from the slower nodes further to the right. If the tree has more than two levels of internal nodes (i.e. the root and direct children), we switch the approach to prioritizing local communication. In this case, internal nodes are assigned to parents that belong to the same δ-agglomerate or, if this is not possible, to the parent with the lowest latency (lines 6– 10 in the Algorithm .4).

Once the configuration of the internal nodes is defined, it remains to allocate the nodes of the remaining groups as leaves of the tree. At this point, we want to maximize local communication, so we allocate the leaf nodes of each group to a parent node that belongs to the same δ-agglomerate or, when this is not possible to the parent that minimizes latency (lines 12– 14 in the Algorithm .4).

Heuristic Using Local-Dissemination The goal is to input restrictions on the environment to achieve the most high-performing tree for this condition. The idea comes from creating subtrees of local dissemination only. Global communication only happens when the roots of each subtree communicate. Having minimized the latency needed by prioritizing the local dissemination, the shape of these subtrees becomes dependent on the bandwidth available. Here, we need the number of internal nodes and how many are from each cluster. The logic from the heuristic Bandwidth-aware distribution can calculate this.

This logic is more formally defined in Algorithm .5

From a group previously defined by the according heuristic previously defined and fixed message size bk, we calculate the maximum fanout wanted for each δ-aglomerate i as so:

$$M_i = \frac{bandwidth_i}{blocksize}$$

This way, for a root of δ-aglomerate i, we configure the internal nodes as follows:

- get one internal node from all other δ-aglomerates and add it as a child (lines 3– 4 in the Algorithm .5).
- get the internal nodes from the δ-aglomerate i (the root one) and create a subtree of maximum fanout M_i from these nodes (lines 6– 10 in the Algorithm .5).
- for each child of the root that is not from the same δ-aglomerate do the same logic and create a subtree for each with the corresponding maximum fanout M_i(lines 10– 16 in the Algorithm .5).

Algorithm .5: Build an informed tree

Input: A list B_i with the group of I internal nodes. C - number of δ-aglomerates. bk - block size.
Output: Tree T.

```
/* Add root to tree                                          */
1  root ← rootHeuristic();
2  T ← {root} ;
   /* Global dissemination on the first
      level                                                  */
3  for internal = 1, 2, ... C do
4      root.children ← root.children ∪ {a
        node from δ-aglomerate C_i not chosen
        from group B_i}
   /* Calculate wanted fanout per
      δ-aglomerate                                           */
5  for i ∈ range(C) do
6      M[i] ← bandwidth(i) / bk;
   /* Local dissemination on the first
      level                                                  */
7  available ← nodes from group B_i from the
    same location of root;
8  while available! = ∅ do
9      for ... M[root.location] do
10         root.children ← root.children ∪ {
            node from available not chosen };
   /* Create the remaining subtrees                          */
11 for internal = 1, 2, ... root.children do
12     if internal.location! = root.location then
13         available ← nodes from group B_i
            from the same location of
            internal.location;
14         while available! = ∅ do
15             for ... M[internal.location] do
16                 internal.children ←
                    internal.children ∪ { node
                    from available not chosen };
   /* Add leafs                                              */
17 for deepestInternal ∈ T do
18     for ... M[deepestInternal.location] do
19         deepestInternal.children ←
            deepestInternal.children ∪ { node
            from remaining nodes not chosen };
20 return T
```

This approach mitigates the bandwidth bottleneck when disseminating within a δ-aglomerate.

For the leaves, we keep the same goal and go to the deepest internals of each subtree and add those, always considering the maximum fanout for the δ-aglomerate the node is in (lines 16– 19 in the Algorithm .5).

4. Results & discussion

In our evaluation, we wish to answer several questions, specifically about the gains and disadvantages of our implemented heuristics when compared with the state of the art. More precisely, we address the following questions:

- What is the impact of our heuristics in heterogeneous deployments?
- Is the sequence of leaders known to an attacker?
- Is the system fair?

To answer the last two, we look at the sequence phase described in Section 3.2.2 and guarantee that by using the VRF the sequence will not be deterministic and an attacker will not be able to exploit. Also, if the function used in VRF we can say that we achieve fairness in the long term.

Next, we will analyze the impact of the heuristics used in the generation phase. The experimental evaluation was conducted using a discrete event simulator in Python already used in other works [10, 11]. The simulated network model follows the latency and bandwidth matrix from ResilientDB [7] and described in Table 2, with the processes distributed evenly across the six data centres. To add the bandwidth model to the existing simulator, we implemented an algorithm that models the flow of messages in a simulation, accounting for bandwidth and latency, to simulate the order and timing of message delivery between nodes. Each node in the simulation maintains sender and receiver channels. When a message is sent from a source node to a destination node, the algorithm calculates transmission times based on message size and bandwidth between them. It schedules message transmission and reception events, considering current time and channel availability to ensure proper sequencing. Messages are processed upon delivery.

4.1. Generic Deployment

We coded the algorithms described in Section 3.3 regarding the generic approach in the simulator and considered three typical scenarios: a scenario in which the *fanout* defined by the administrator coincides with the number of δ-agglomerates (Scenario 1), a scenario in which the *fanout* is higher (Scenario 2) and another in which it is lower (Scenario 3). As the main metric, we used the time needed for the root to collect the quorum, which is the instant from which the consensus algorithm can make progress.

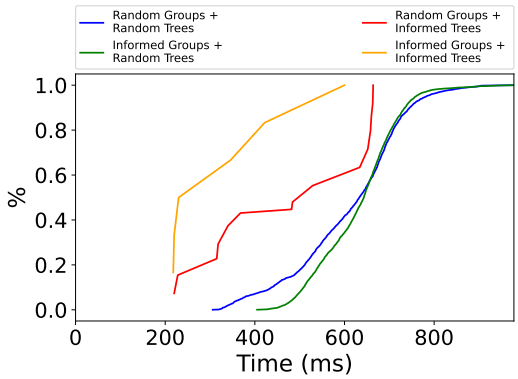


Figure 1: Time to retrieve quorum in Scenario 1.

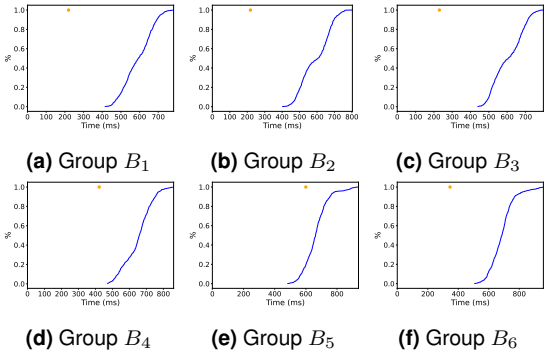
To independently assess the impact of the heuristic for allocating nodes to groups and building trees from a group, we evaluated the following combinations for each scenario: i) informed groups with informed trees; ii) informed groups with random trees; iii) random groups with informed trees; iv) random groups with random trees.

Scenario 1: M = number of δ-agglomerates

With 6 δ-agglomerates for this scenario, we set the *fanout* $M = 6$ and create a regular two-level tree. For this purpose, we have $N = 43$ nodes. We generated ten different groups for the random groups, and for the random trees, we generated 100 different trees for each group. The results for this scenario in the four combinations are shown in Figure 1.

Starting with the construction of random trees (green and blue lines), you can see that they perform worse than informed solutions. The differences between building random trees from a random or informed group distribution are not significant and, in the case of this figure, the difference between the results is due to the fact that we weren't able to take a large enough sample. In fact, as you'll see below, these lines appear increasingly overlapping in the other scenarios. On the other hand, the informed algorithm for building trees already shows significant gains, even when the constitution of the groups is still random; in this case, we see an increase in performance by 30% in half of the trees. The informed distribution of the nodes among the groups improves the results even more (in the order of 70%), as it increases the probability of there being a well-situated root in all the groups.

Figure 2 shows the impact of combining the different heuristics. In particular, we selected a dis-



(a) Group B_1 **(b)** Group B_2 **(c)** Group B_3

(d) Group B_4 **(e)** Group B_5 **(f)** Group B_6

Figure 2: Informed heuristic (yellow) vs random strategy (blue) in Scenario 1.

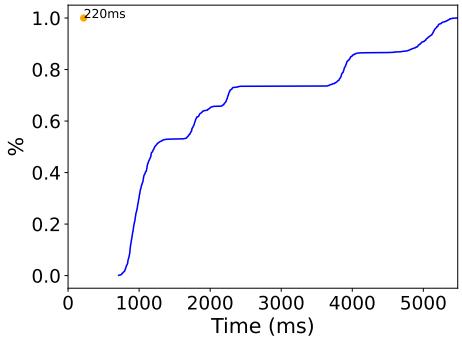


Figure 3: Informed heuristic (yellow) vs random strategy (blue).

tribution of nodes by groups and compared the informed tree that is built by our heuristic from each group (yellow dot), with several random trees that can be built from the same group (blue line). As you can see, for most groups, the informed tree performs better than any of the 100 randomly constructed trees, which shows that the probability of obtaining a good tree in an uninformed way is generally low. However, the figure also shows that our heuristic does not always succeed in obtaining the best tree. In Figure 2e, for example, group B_5 includes 2 nodes in Taiwan and 1 node in each of the other δ-agglomerates. Although the δ-agglomerate with the lowest average latency for the other δ-agglomerates in this scenario is in Iowa, finding a more efficient tree with the root in Taiwan is possible.

4.2. High-performing Deployment

We coded the algorithms described in Section 3.3 regarding the heuristic using the local-dissemination approach. The same simulator, metrics and informed group distribution were used with the nodes uniformly distributed between the 6 data centres. For 430 nodes, messages of size 2000 Mbit/s and the Bandwidth-aware Distribution of nodes in groups described in Section .3. The proposed solution, where we fully leverage the local dissemination and create subtrees with fanout based on the bandwidth bottleneck of each cluster from a root from the cluster with the most bandwidth, was compared with a random distribution approach where we create 1000 random trees as follows: we connect the root to the internal nodes in the group and for the leaves, we randomly distribute them by a fixed fanout.

We gathered the results for a group in Figure 3. As expected from the previous heuristic results, the informed heuristic outperforms the random approach. For half the random trees, the informed heuristic can outperform the other by 80%. Even though the testbed of 1000 random trees could be further improved, it seems that in this heuristic, it will be much harder to find cases like in Figure 2e) where we have random trees with better performance than those found by our heuristic.

4.3. Discussion

For the generic deployment, the evaluation considers a relatively small set of scenarios. Still, it covers different relationships between the number of δ-agglomerates and the fanout of the tree, i.e. scenarios in which the fanout is equal (Scenario 1), higher (Scenario 2) or lower (Scenario 3) than the number of δ-agglomerates. In all scenarios, our generic tree-building heuristic achieves, on average, much shorter collection times than the random trees currently used by Kauri. Furthermore, in all cases, the informed division of nodes into groups improves the results of the heuristic used to build a tree in an informed way. This effect is less pronounced in Scenario 3. Still, more experiments would be needed to understand whether it is the value of the fanout or the depth of the tree that most affects the impact of the informed distribution of nodes by groups. The evaluation also shows clusters for which it is easy to find, even at random, trees with better performance than those found by our heuristic (for example, the case presented in Figure 2e). We are investigating whether it is possible to improve the generic heuristic to avoid these cases without making it unnecessarily complicated (an advantage of the current version is that it is very efficient to run from a computational point of view).

For the last deployment, we still registered a significant improvement in collection times than random trees. Even though we have to expand the experiments to understand the impact with different bandwidth models and message sizes, the scenario indicates we can find a tree better than the random best case in a determinist and computationally efficient manner.

5. Conclusions and Future Work

Most permissioned blockchains are based on variants of BFT consensus protocols that have scalability problems since all participants engage in mul-

tiple rounds of data exchange. Kauri [12] combines dissemination/aggregation trees and pipelining to distribute the load and compensate for the increased latency of using trees. However, it uses a stable-leader approach to leverage the benefits of pipelining. Several arguments favor changing the leader between consecutive consensus rounds to provide censorship resistance. These arguments motivate the need for a strategy to rotate the leader with a minimal negative impact on pipelining and promoting a good load balance among all participants.

We surveyed the state-of-the-art BFT algorithms. We analyzed their communication patterns and discussed leader rotation strategies, the impact of the pipelining techniques and how they behave in heterogeneous deployments.

In this work, we present a solution that aims to enrich Kauri with heuristics for generating dissemination and aggregation trees based on the network topology. In this context, we propose a heuristic for building trees that uses information about the latency between nodes to create groups and another for generating trees from these groups. We evaluate both heuristics using simulations based on realistic network latencies and present results that suggest it is possible to reduce the time needed to collect a Byzantine quorum by 70%. We elaborated a solution to support a rotating-leader approach in a tree topology that leverages these heuristics while maintaining an acceptable throughput by the pipelining technique.

In future work, we intend to evaluate our approach in a real code scenario running in geo-distributed data centres. Another relevant scenario would leverage pipelining techniques to assess how this rotating strategy behaves. In the generic heuristic, we also consider that the system administrator provides the M and δ parameters; it would be interesting to extend the work to configure these parameters automatically. As mentioned in the evaluation chapter **??**, the impact of increasing the depth of the trees when leveraging these heuristics needs to be further studied to understand if it is the fanout of the tree or its depth that has the most impact. Also, for the clusters where it is easy to find a random tree that outperforms one using our heuristic, we need to investigate whether it is possible to improve our heuristic to avoid those cases without incurring a much higher computational cost. Lastly, in our last heuristic, we must expand the testbed to include different bandwidth models and message sizes in real-world scenarios.

References

[1] R. Albert, H. Jeong, and A.-L. Barabási. Diameter of the world-wide web. *nature*, 401(6749):130–131, 1999.

[2] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 197–206. IEEE, 2008.

[3] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[4] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, volume 9, pages 153–168, 2009.

[5] S. Cohen, R. Gelashvili, L. K. Kogias, Z. Li, D. Malkhi, A. Sonnino, and A. Spiegelman. Be aware of your leaders. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*, pages 279–295. Springer, 2022.

[6] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

[7] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi. Resilientdb: Global scale resilient blockchain fabric. *arXiv preprint arXiv:2002.00160*, 2020.

[8] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th usenix security symposium (usenix security 16)*, pages 279–296, 2016.

[9] P. Li, G. Wang, X. Chen, F. Long, and W. Xu. Gosig: a scalable and high-performance byzantine consensus for consortium blockchains. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 223–237, 2020.

[10] M. Matos, P. Felber, R. Oliveira, J. O. Pereira, and E. Rivière. Scaling up publish/subscribe overlays using interest correlation for link sharing. *IEEE Transactions on Parallel and Distributed Systems*, 24(12):2462–2471, 2013.

[11] M. Matos, H. Mercier, P. Felber, R. Oliveira, and J. Pereira. Epto: An epidemic total order algorithm for large-scale distributed systems. In *Proceedings of the 16th Annual Middleware*

Conference, Middleware '15, page 100–111, New York, NY, USA, 2015. Association for Computing Machinery.

[12] R. Neiheiser, M. Matos, and L. Rodrigues. Kauri: Scalable BFT consensus with pipelined tree-based dissemination and aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 35–48, 2021.

[13] S. E. Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[14] J. Seibert, S. Becker, C. Nita-Rotaru, and R. State. Newton: Securing virtual coordinates by enforcing physical laws. *IEEE/ACM Transactions on Networking*, 22(3):798–811, 2014.

[15] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 135–144. IEEE, 2009.

[16] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. Hotstuff: Bft consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069*, 2018.

[17] G. Zhang and H.-A. Jacobsen. Prosecutor: An efficient bft consensus algorithm with behavior-aware penalization against byzantine attacks. In *Proceedings of the 22nd International Middleware Conference*, pages 52–63, 2021.