BFTree - Scaling HotStuff to Millions of Validators

Jason Ansel Marek Olszewski

Celo.org

DRAFT VERSION 0.11

Abstract

A large selling point of Nakamoto consensus is that it can scale elastically with the number of untrusted participants working together to achieve consensus. This is a big advantage with regards to decentralization that has evaded many cryptocurrencies using proof-of-stake consensus protocols to-date, especially ones based on Byzantine Fault Tolerance (BFT) consensus. This paper outlines a novel modification to BFT consensus, named BFTree, that increases the practical number of validators that can be used in a BFT system from hundreds to millions of validators, without the use of sharding or subcommittee sampling. BFTree arranges validators into a virtual tree, to parallelize signature aggregation between non-byzantine nodes working to achieve consensus. When byzantine nodes interfere with the aggregation, the roots of all subtrees that were able to achieve agreement perform BFT consensus to finish the round, frequently with fewer messages than if all validators participated. By thoughtfully reorganizing the tree such that nodes that have historically been reliable are paired with other reliable nodes, BFTree limits the impact that a byzantine node can have. This organization strategy allows an honest and reliable quorum of validators to quickly aggregate the required number of signatures in a distributed manner, allowing the algorithm to scale to large numbers of validators.

1 Introduction

There has been a growing number of cryptocurrencies using proof-of-stake consensus protocols [7, 8, 12, 18] that are based on or are moving to use Byzantine Fault Tolerance (BFT) consensus [13]. These algorithms provide strong finality guarantees in that either all honest nodes will adopt a block or none will, thus eliminating the possibility of forks and rollbacks found in protocols based on Nakamoto consensus [15]. Unfortunately, existing BFT algorithms do not scale well. Most actively used implementations scale to 100's of participants, with some upcoming systems aiming to scale to 1000's of participants with some sacrifices (e.g. increased block times).

This has led to many modern cryptocurrencies [8, 12] adopting a two-class system where there is a smaller set of distinguished nodes that act as *validators* and participate in the BFT algorithm while the common node is merely an observer and does not participate. This design creates many complexities in terms of rewards, and either leads to explicitly designed or emergent *delegation* schemes that allow many smaller nodes to pool their stake in order to participate in validation and thus get rewards. The incentives of this delegation can be tricky to get right and may lead to people getting exploited by dishonest middlemen, or a centralization of control of the network in the hands of a small number of delegation pools.

This paper presents a novel modification to BFT algorithms called BFTree, designed with the goal of making BFT consensus scale to millions of validators. This change can enable a more decentralized proof-of-stake protocol by eliminating the need for two classes of nodes and delegation to a small number of validators. We view the goal of scaling to millions of validators as a forcing function to create a more scalable consensus algorithm where BFT is no longer the bottleneck in terms of making cryptocurrencies more decentralized. In practice, other bottlenecks, such as block sizes, may make the ideal number of validators be tens or hundreds of thousands of nodes for large cryptocurrencies.

To understand what will make BFTree scale, first one should understand the bottleneck in current BFT algorithms: the communication pattern. pBFT [4] and most widespread variants of BFT algorithms (e.g. [3]) require all-to-all communication. This all-to-all communication requires $O(n^2)$

messages and in the naive implementation each message is $O(n)$ in size (though this can be optimized). This means that thousands of validators will generate millions of messages, which in practice becomes the limiting factor. A more recent algorithm, HotStuff BFT [20], which has shown a lot of promise but is not yet deployed in a live cryptocurrency, improves this communication pattern to all-to-one, requiring $O(n)$ messages on the critical path. This will allow BFT to scale another order of magnitude over pBFT, but all-to-one will not scale to millions of validators, since validating a million signatures on a desktop CPU can takes minutes of time, and it is not feasible without distributing the work across a number of machines.

BFTree arranges the validator communication into a dynamically constructed tree structure, thus requiring $O(log(n))$ parallel steps to aggregate the tree in the best case (see Figure 1 for an example). The message size is fixed, and signatures are combined in a distributed way as they work their way up the tree. Nodes in the tree operate by unanimous consent and if there is disagreement (or if a node is offline) at any branch in the tree the two subtrees split and operate independently as virtual validators. At the end of this tree consolidation process there will be a much smaller number of virtual validators each representing a larger subtree of nodes. This set of virtual validators then run an existing BFT algorithm, such as HotStuff BFT, to arrive at a final consensus.

This tree structure gives us a best case of $O(log(n)^2)$ messages per block (O(log(n)) broadcasts are needed to verify each parties behavior), but there is the problem that Byzantine nodes positioned strategically throughout the tree could target the network and slow things down. We address this problem with a key concept in BFTree: position in tree is determined by past reliability.

The leftmost node in the tree is the most reliable with the longest history of good behavior (being part of the consensus) while the rightmost nodes are those nodes with recent failures or new nodes joining the network. Over time this will cause a sorting of the tree, and while a Byzantine node could moderately slow down a single block, they could only do that once before being repositioned in the tree and losing the power to repeat the bad behavior. If one assumes a fixed set of Byzantine nodes (<1/3rd of the total), eventually each of them will act in a Byzantine way and the leftmost two thirds of the tree will be only correctly operating nodes, at which point the best case will happen for every subsequent block regardless of the behavior of Byzantine nodes.

2 Related Work

A related technique to make blockchains scale is sharding [5, 14]. Sharding is especially critical for scaling the number of transactions per seconds in a network. When sharding BFT algorithms, there is a risk of reduced security as the attacker only needs to compromise a quorum of a shard rather than than all validators. OmniLeder [11] proposes novel techniques for mitigating these security losses from sharding. Sharding can be combined with the techniques described in this paper to allow more validators per shard and have a network that scales both in the number of transactions and in the number of validators.

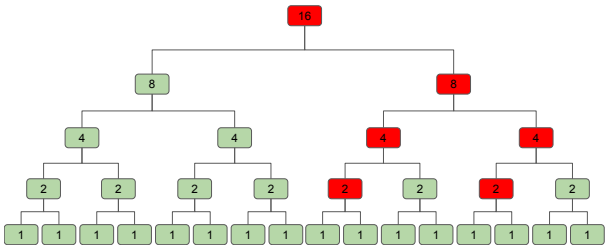
An alternate approach relies on committee sampling to lower the number of validators used to validate any single block at the expense of introducing weaker probabilistic guarantees [7, 9]. Such designs, and other probabilistic PoS protocols (e.g. [16]) lose the strong theoretical guarantees provided by BFT, and are therefore harder to reason about with regards to attack resilience, especially in cases where the committee members change infrequently, or can be predicted or manipulated.

Threshold signatures [1, 2, 6] provide cryptographic primitives that allow one to verify that at least N of M parties have signed a block without revealing which parties signed the message. Unfortunately, these systems require a distributed setup phase where a threshold public key is generated. Requiring this setup phase would not be practical in a system with millions of validators where one would want to support validators joining and leaving the network. Instead BFTree only using the simpler non-threshold BLS signatures (N of N) so that it can support frequent changes to the validator set and no setup phase is needed.

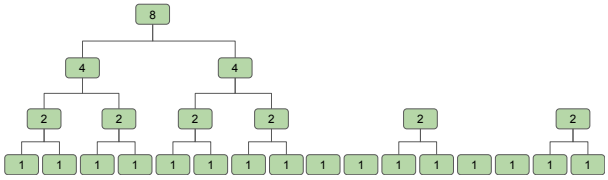
ByzCoin/CoSi [10, 17] also use communication trees for signature aggregation. They scale the transaction rate of BitCoin by collecting many *witnesses* for statement using this tree structure. ByzCoin does not replace the proof-of-work structure of BitCoin and still requires miners with hashing power. The main technical differences with BFTree are the added reputation system in BFTree and the ability of BFTree to verify the work of the tree level-by-level, thus avoiding the fallback to a flat communication structure in ByzCoin.

3 HotStuff BFT

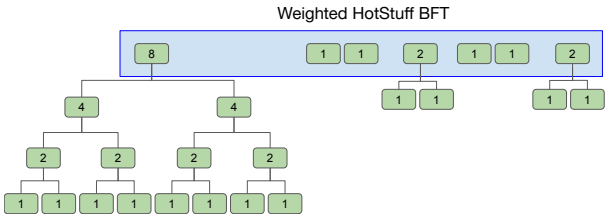
This section provides a brief overview of HotStuff BFT. See the original HotStuff BFT paper [19, 20] for more thorough



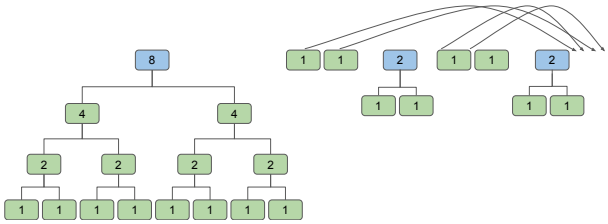
(a) Validators start by communicating with their siblings in the tree to aggregate consensus. Red nodes show places where the children did not agree with each other.



(b) Next, the tree is pruned of any red nodes, so that only subtrees that have achieved consensus remain.



(c) The root nodes of all remaining subtrees perform weighted HotStuff BFT consensus.



(d) Before the next round, faulty nodes are grouped together and moved to the right side of the tree.

Figure 1: A step by step example of how BFTree achieves consensus in the presence of faulty validators.

explanation and proofs of correctness.

The fundamental mechanism in HotStuff is the counting of the votes of at least $\frac{2}{3}n+1$ out of n nodes. This voting process repeats over and over, once per block. Each block has a pointer to the last valid block, and validators decide how to vote using the SafeNode and transaction locking rules defined in [19]. The consensus for a transaction is chained over four total blocks. HotStuff pipelines this process so that, assuming no failures, a single block k containing one set of vote results will be:

- The *prepare* phase for block k (itself)
- The *pre-commit* phase for block $k-1$
- The *commit* phase for block $k-2$
- The *decide* phase for block $k-3$, these transactions are now final.

It is possible for there to be forks less than three in length, but once a block is four levels deep, it is locked, and one is guaranteed that a quorum of nodes has finalized the transactions in the block and will never vote for a chain not containing those transactions. The intuition is similar to that of other BFT algorithms. In the first vote a quorum agrees. In the second vote a quorum knows a quorum agrees. In the third vote a quorum knows that a quorum knows a quorum agrees, and that a quorum has finalized those transactions so no future vote can reverse those transactions. By pipelining these steps across multiple blocks and adding an extra phase, HotStuff eliminates the costly $O(n^2)$ communication found in other pBFT protocols.

4 Assumptions and Requirements

BFTree relies on an underlying signature algorithm that supports the combination of both signatures and public keys into aggregate signatures and public keys that use the same space as a single signature. We assume the following methods:

- keygen() => pk1, secret1
- sign(message, secret1) => sig1
- verify(message, pk1, sig1) => boolean
- combine_public_keys(pk1, pk2) => pk3
- combine_signatures(sig1, sig2) => sig3

The signature scheme should have the property that: `verify(message, combine_public_key(pk1, pk2), combine_signature(sig1, sig2)) == True` Which implies: `verify(message, pk1, sig1) == True` and `verify(message, pk2, sig2) == True`. If either of those is false, the combined verification should fail. This property must apply recursively so it can be used to combine an arbitrary number of signatures into a single signature. An example of such a signature scheme is BLS signatures [2].

We also rely on an underlying (less scalable) BFT algorithm which we call at the end with the top of each subtree as virtual validators. This underlying BFT algorithm must support weighted votes (weighted by the total stake of the subtree) and allow the set of virtual validators to change over time. For the remainder of the paper, we will assume this algorithm is HotStuff BFT, which can be easily modified to have these properties, but in principle any BFT algorithm would work. HotStuff only requires a single vote counting phase, to use this with pBFT one would need to run the vote counting process multiple times for each phase in pBFT.

We also make assumptions about network latency. To make forward progress (arrive at consensus) at least 2/3rds+1 of nodes must be both correctly behaving and have a bounded maximum network latency to talk point to point to each other. If this property is violated the system fails by halting progress and minting no valid blocks until network is restored. This network latency upper bound can be set dynamically where it is increased when consensus is not reached and decreased when consensus is reached especially quickly. We assume the existence of a O(log(n)) broadcast operation, implemented using standard algorithms, and we also assume that the point to point communication is validated

5 BFTree Algorithm

We will first describe the high level phases in the algorithm, and then go into detail about the algorithm that each actor uses.

5.1 High Level Phases

- **Phase 0:**

  The block producer (chosen through an algorithm such as weighted round robin) generates a candidate block and broadcasts that block along with a signed tuple of:

  block_info = (block_hash, prior_block_depth, current_block_depth)

  to all nodes. Each node validates that the block is valid (according to the HotStuff BFT rules), and that the depths and signatures of the block producer are valid. If so, they sign block_info, if not they sign (null, current_block_depth) to indicate a no vote.

- **Phases 1 through ceil(log(n)):**

  Each level of the tree in bottom up order communicates with their neighboring branch to check if they agree. Agreeing signatures are shared and combined and the left peer operates the virtual node one level up in the tree. Disagreeing nodes (or the root node) forward (block_info, signature, first_index, last_index) to the block producer. Where first and last index represents the range of the subtree the signature includes.

- **Phases ceil(log(n))+1 to 2*ceil(log(n)):**

  In each of these phases the block producer broadcasts the partial signatures received so far. These broadcasts are a single aggregated signature and a list of ranges indicating the subtrees included.

  Each phase represents a level of the tree in top down order (the opposite order as before). In the corresponding phase the right side branches check the work of the left branches, they are expecting to see their combined signature included broadcast ranges. If they do not, they forward (block_info, signature, first_index, last_index) to the block producer.

- **Phase 2*ceil(log(n)) + 1:**

  In the final phase, the block producer broadcasts the final block and signatures to everyone.

5.2 Node Algorithm

Each individual node runs the following algorithm. The inputs are a secret key for current node, current_block_depth, and index in the tree.

```
# Receive the candidate block info from the block producer
block_info_candidate = receive_from_block_producer()
(block_hash, prior_block_depth, proposed_current_block_depth) = block_info_candidate

# Verify the block info candidate using HotStuff and chain rules.
if verify_block(block_info_candidate, ...)
  block_info = block_info_candidate
```

```
else:
  block_info = (None, current_block_depth)

signature = sign(block_info, secret)

start_index = index
end_index = index
my_level = TREE_DEPTH

# Combine signatures with siblings in the tree
for level_from_bottom in range(TREE_DEPTH):
  # Once we've reached the top of the tree, send the aggregated signatures to the block producer.
  if level_from_bottom == TREE_DEPTH - 1:
    send_to_block_producer(block_info, start_index, end_index, signature)
    my_level = level_from_bottom
    break

  # Check if this node is a virtual node at the current tree level
  if (index % 2**(level_from_bottom + 1)) == 0:
    # This node is a virtual node, so aggregate itself with its sibling
    sibling_index = index + 2**level_from_bottom

    # Share block info with sibling at the current level of the tree
    (block_info2, start_index2, end_index2, signature2) = receive_from_node(sibling_index)
    send_to_node(sibling_index, block_info, start_index, end_index, signature)

    # Check if there is agreement between both siblings and if not, short circuit by messaging the block producer directly.
    if verify_peer(sibling_index, block_info2, start_index2,
                   end_index2, signature2, ...):
      end_index = end_index2
      signature = combine_signatures(signature, signature2)
      # Continue one level up tree
    else:
      send_to_block_producer(block_info, start_index, end_index, signature)
      my_level = level_from_bottom
      break
  else:
    # This node is not a virtual node, so aggregate with its sibling but don't continue up the tree
    sibling_index = index - 2**level_from_bottom

    # Share block info with your sibling at the current level of the tree
    send_to_node(sibling_index, block_info, start_index, end_index, signature)
    (block_info2, start_index2, end_index2, signature2) = receive_from_node(sibling_index)

    # Check if there is agreement between both siblings and if not, message the block producer directly since the sibling will likely not propagate this node's vote.
    if verify_peer(sibling_index, block_info2, start_index2,
                   end_index2, signature2, ...):
      # Common case, don't send anything
    else:
      send_to_block_producer(block_info, start_index, end_index, signature)

    my_level = level_from_bottom
    break

# Check that this node's signature made it into the block, and if not, send it directly to the block producer
for level_from_bottom in reversed(range(TREE_DEPTH)):
  if my_level >= level_from_bottom:
    # ranges is a list of (start, end) tuples
    (block_info2, signature2, ranges) = receive_broadcast_from_block_producer()

    # Check if my info is missing from broadcasted ranges
    if not ranges_contains(index, ranges) and block_info == block_info2:
      send_to_block_producer(block_info, start_index, end_index, signature)
    else:
      break

final_block = receive_final_block_broadcast_from_block_producer()
```

Each `send` and `receive` operation should verify the identity of the sender with a signature and have a timeout based on the phase of the algorithm. For example the 5th possible receive should timeout at time 5*k after the algorithm begins. Timed out receives should return null and fail validation. Each point to point send is allocated a timeout of k, and each broadcast is allocated a timeout of k*log(n). Timeouts in loop phases skipped with conditions or breaks should be counted. So the cumulative timeouts of the first loop are k*TREE_DEPTH and the cumulative timeouts of the second loop are k*log(n)*TREE_DEPTH. The timeout scaling factor, k, should be set dynamically across blocks. Failed consensus results in increasing k by a scaling factor (up to some hard maximum), and especially fast consensus results in decreasing k by a scaling factor.

5.3 Block Producer Algorithm

First the block producer creates the block according to Hot-Stuff BFT rules. Note this typically means there just a pointer to the previous block as the algorithm involves voting on the prior block not the current block (the transactions for which the block producer can gather asynchronously and publish in the last step.

```
ranges = []
signature = None
finished = False
block_info = (block_hash, prior_block_depth,
 ↪  current_block_depth)

# Broadcast the candidate block info
broadcast(block_info)


def thread1():
  while not finished:
    # Process received messages with a priority
     ↪  queue (based on total stake):
    (block_info2, start_index, end_index, signature2) =
     ↪  receive_by_priority()

    # Validate the received signatures
    if (block_info == block_info2 and
          valid_new_ranges([start_index, end_index],
           ↪  ranges)
          verify(block_info, get_public_key(start_index,
           ↪  end_index), signature2):
      new_ranges = append_and_merge(ranges, [start_index,
       ↪  end_index])
      new_signatures = combine_signatures(signature,
       ↪  signature2)
      with lock:
        ranges = new_ranges
        signature = new_signatures


def thread2():
  for level in range(TREE_DEPTH):
    wait_for_phase(ceil(log2(num_validators)) + level + 1)
    with lock:
      broadcast(block_info, signature, ranges)


# Start threads
t1 = threading.Thread(target=thread1)
t2 = threading.Thread(target=thread2)
t1.start()
t2.start()

wait_for_phase(2 * ceil(log2(num_validators)) + 1)

finished = True

# Cause receive_by_priority() to exit with Nones
terminate_receive_by_priority_queue()

# Stop and join the two threads
t1.join()
t2.join()

# Broadcast the final block, ranges, and
 ↪  signatures.
broadcast(block_info, ranges, signature)
```

5.4 Tree Reshuffling

Between each block (or n blocks), the tree is changed with the goal of moving more reliable nodes left and less reliable nodes right. Tree reshuffling is run on every node and the result included in the block in the form of a merkle tree. Reshuffling is done by looking over a sliding window of the last W valid blocks and counting how many infractions each validator has where an infraction is either:

1) Not being part of a consensus

2) Two nodes that should have combined signatures in a tree instead send their signatures to the block producer directly. (The block producer includes a proof of this in the form of the two signed messages in the block body.)

Each of these infractions is given a different weight, then the infraction count is summed up for each validator over the sliding window. The top R validators with non-zero infraction counts sorted by infraction count descending and tree position are removed from the tree and re-inserted at the right side of tree in a random order.

This has the effect of creating large subtrees on the left that consist of the most reliable nodes and the unreliable nodes will be on the right side of the tree. The block producer can then preferentially chose to process messages coming from the left side of the tree where a 2/3rds+1 vote can be accumulated quickly in just a few messages.

This tree reshuffling phase also includes both adding and removing validators from the tree. New validators are added at the rightmost side and retiring validators are removed.

5.5 Signature Verification

These algorithms require the ability to get the combined public key of all nodes between a given start_index and end_index quickly. While this naively takes O(n) work, by precomputing the tree of public keys (and including this as merkle tree in the

block) one can reduce the cost of generating these public keys and verifying signatures to O(log(n)) for an arbitrary range and constant time for the branches in the tree. The cost of initially generating this public key tree can be amortized over many blocks, and in our tests an unoptimized implementation on a Core i9 processor took under three minutes to generate a tree for one million validators. Updating a public key tree after a reshuffle can be done in O(r*log(n)) time, where r is the number of nodes moved. There is also a lot of room for optimization here, as any one node only needs a small subset of the tree and the tree need not always be perfectly balanced.

6 Optimizations and performance.

This basic algorithm can be optimized for speed to reduce the total number of phases required.

1) Increase the branching factor to be larger than 2.
2) Remove the top few levels of the tree (which are unlikely to agree) and skip their phases.
3) Have the block producer end the algorithm without waiting for stragglers if 2/3rds+1 votes are collected. (We recommend block rewards give a small bonus for oversigning a block to balance this.) Potentially most of the broadcast phases could be skipped in the common case.
4) The block producer's thread1() could be parallelized further.

Naively one million validators would take 42 total phases to complete, of which 22 would require a broadcast. Using a 32-way branching factor would reduce that to 10 phases (6 broadcasts), then removing the top 2 levels of the tree would further reduce this to 6 phases (4 broadcasts). 32-way branching would work by having all nodes send their signatures to the leftmost node in the branch, then that leftmost node would internally simulate the combining logic in the binary tree. Finding the optimal branching factor would require performance testing.

The absolute performance would depend a lot on what the network timeout is set to. In a cryptocurrencies that slashes validators for not being part consensus repeatedly (being offline) there is a strong incentive to run validators in data centers and well connected networks. A validator with a network far slower than the 2/3rd+1th percentile of other validators may get slashed for being too slow to be included in consensus for a series of blocks. We show results from detailed simulations in section 10.2.

7 Correctness

Theorem 1. *For a sufficiency large network timeout K, BFTree will reach consensus if 2/3rds+1 of nodes and the block producer are honest and vote for the block.*

Proof. Set the network timeout K such that no messages from functional nodes timeout and all messages to the block producer are processed. By contradiction, assume that consensus is not reached. Then there must exist some node Q that is honest and votes for the block, but is not included in the tallied votes by the block producer. Then in the second loop of the algorithm, on line 47, Q would send its vote to the block producer and that would have been included, resulting in a contradiction.

Following the rules of HotStuff BFT (and the corresponding proofs), this correct vote tallying property in Theorem 1 establishes that BFTree is no worse than HotStuff BFT in terms of correctness and given a large enough timeout.

□

8 Tree Sortedness

Much of the performance of the algorithm will depend on the ability of the tree to sort the reliable and unreliable nodes correctly. It is based on an intuition that future reliability of nodes is correlated to past reliability of nodes. In the extreme case where a set of nodes A is perfectly reliable and another set of nodes B is Byzantine, one would expect the tree to eventually become sorted as only the Byzantine nodes would incur infractions that reinsert them into the tree to the right and correct nodes would not. At the other extreme, if Byzantine behavior is uncorrelated between blocks and a random set of nodes act out each block, then this tree based reputation score would be defeated. We believe that in reality we will be closer to this first scenario than the latter.

9 Potential Attacks

One attack is a network based attack where the block producer (or some node) is flooded with fraudulent messages that prevent legitimate messages from being processed due to CPU or network bottlenecks. This attack is not unique to BFTree, and it is something HotStuff and others are also vulnerable to. One possible mitigation is to try to keep the network addresses of participants secret using techniques such as an onion router. This unfortunately has high overheads, so

it may not be practical. The mitigation we suggest is processing messages in a priority queue order, where lower priority messages are either ignored or blocked by a firewall. The block producer can prioritize messages by total stake size and position in the tree, which can be verified through signatures. Individual nodes communicate point to point where they can verify the public key of the message sender. One could imagine adding a network address reputation system on top of this where participants track the addresses generating both valid and invalid messages and then prioritize checking signatures of messages from addresses that generate the highest fraction of valid messages.

Another attack vector involves a Byzantine node trying to manipulate the tree structure. This can be done with a type 2 infraction, where a node refuses to combine signatures with their partner in the tree and both sides send their signature directly to the block producer. In this case, both the Byzantine node and their partner in the tree would incur the same infraction points so there is a cost to this type of manipulation. This type of manipulation is also limited by tree position. After doing this once, a Byzantine node would be pushed to the right of the tree where they would only be partnered with other unreliable nodes and could not affect the reliable nodes on the left side of the tree. Concerns about this type of manipulation are what motivated randomizing the order that misbehaving nodes are added to the right side of the tree. This randomization makes is harder for a possible attacker to control who they are paired with. We have not yet come up with an effective way for an attacker to profit from this manipulation.

10 Simulated Performance

In this section, we share a number of findings from detailed simulations of the BFTree protocol.

10.1 Impact of Byzantine Nodes

The block producer will quickly become the bottleneck as failures spread throughout the tree since failures will result in validators sending signatures directly to the block producer, who will then have to aggregate the signatures themselves. Figure 2 shows simulated results for how many messages a block producer will need to process, at various levels of validator reliability. In the worst case of 0% past downtime correlation, we draw a random set of nodes to be down for each block with a binomial distribution, such that downtime

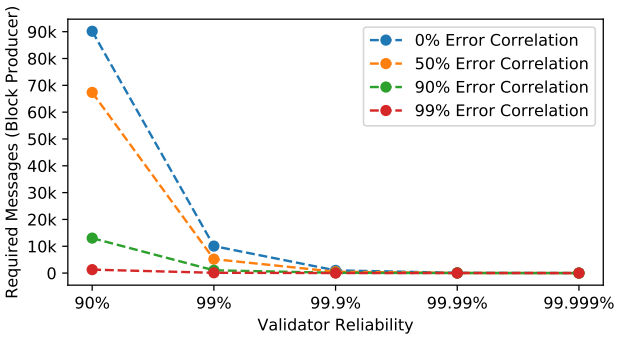


Figure 2: Simulated count of block producer messages required to reach the $\frac{2}{3}+1$ vote threshold with $1,000,000$ total validators each with equal stake. Aggregate validator reliably is the fraction of validators operating correctly in the average block. Error correlation is the correlation of current downtime to past downtime or the chance that a byzantine validator in the current block was also acting maliciously in the prior block.

is not correlated with past performance. Without correlated downtime, tree reorganization provides no benefit. The other lines in the graph show increasingly correlated downtime, where if is node down in block N there is a 50, 90, or 99% chance that it was also down in block $N-1$. We observe a clear benefit where as the correlation of error increases, the message count drops.

The raw performance and block rate depends on how many messages per second the block producer can process. Our testing shows the single-threaded BLS12-381 signature validation/merging can run at approximately $5,000$ signatures per second on a Core i7-8086K processor, however actual performance will vary greatly with implementation, network overheads, and optimization level. In a very rough estimate, these results indicate that consensus can be achieved in a number of seconds in most cases, assuming the network exhibits more than 90% reliability with relatively low error correlation. Encouraging such availability can be achieved by instituting liveness incentives or even slashing conditions such as the ones used by Cosmos [12].

10.2 Signature Aggregation Performance

Figure 3 shows the simulated performance of aggregating signatures using latencies typical in a worldwide network (300ms mean with a 100ms standard deviation and 50ms minimum). In the expected case where unreliable nodes are

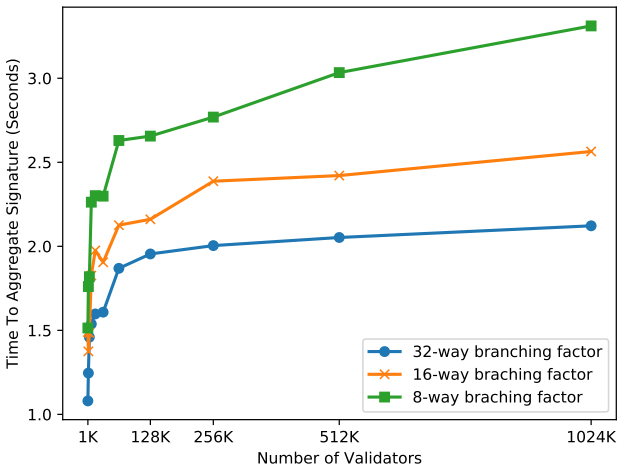


Figure 3: Simulated latency for aggregating signatures by number of validators participating in the tree, shown for various tree branching factors.

grouped together, TreeBFT can achieve consensus with just one broadcast and one tree aggregation, showing that consensus can be achieved in under 3 seconds when using a 32-way tree. Note, however, that we are excluding the time needed for block propagation and validation, and so a realistic block period would be larger. Further testing is required to measure real world performance.

11 Future Work

This algorithm is currently just a concept represented by this paper, slide decks, and many discussions. It is still far from being ready for adoption and we are releasing this in the hopes that attention of the community and vetting by experts will help harden the idea. The remaining work can be divided into more theoretical analysis, simulations, and prototype implementations.

For theoretical analysis, we hope to find any flaws or new attack vectors in this system. We would also like help coming up with correctness and liveness proofs that make weaker assumptions about the network. There is also room for more statistical analysis of how the tree sortedness is affected by different distributions of Byzantine node and other failures.

For simulations, this algorithm (and the optimized version) include a number of parameters that need to be set. We plan to continue expanding our simulations to help us fine tune these parameters. One could also use a simulation to model a wider set of malicious behaviors to see how they affect the performance of the algorithm.

Finally, a healthy consensus algorithm should have many independent implementations. We plan to implement a prototype for testing, but invite others to create independent implementations.

12 Conclusions

Part of what made the idea of Bitcoin so compelling when it launched was that anyone could participate by running a miner in their home. This permissionless structure is a fundamental part of the cryptocurrency revolution. Unfortunately, as we transition to more environmentally sustainable proof-of-stake systems, a piece of this permissionlessness might be lost with the latest two-tier staker/delegator systems that only allow hundreds of people actually to participate in the consensus protocol due to scalability limitations. While some people might prefer to be delegators, we believe that a permissionless protocol is more resilient if it allows anyone to join in the consensus protocol. BFTree takes an important first step in enabling a protocol that can scale to millions of validators in a PoS protocol. We hope that it enables future cryptocurrencies to maintain this compelling property.

References

[1] Dan Boneh, Manu Drijvers, and Gregory Neven. Compact multi-signatures for smaller blockchains. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 435–464. Springer, 2018.

[2] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '01, pages 514–532, Berlin, Heidelberg, 2001. Springer-Verlag.

[3] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains, Jun 2016. Accessed: 2017-02-06.

[4] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[5] George Danezis and Sarah Meiklejohn. Centrally banked cryptocurrencies. *arXiv preprint arXiv:1505.06895*, 2015.

[6] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20:51–83, 2006.

[7] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM.

[8] L.M Goodman. Tezos - a self-amending crypto-ledger white paper, https://tezos.com/static/white_paper-2dc8c02267a8fb86bd67a108199441bf.pdf.

[9] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.

[10] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 279–296, 2016.

[11] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[12] Jae Kwon and Ethan Buchman. Cosmos: A network of distributed ledgers, https://github.com/cosmos/cosmos/blob/master/white paper.md.

[13] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[14] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM, 2016.

[15] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf.

[16] Team Rocket. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies, https://ipfs.io/ipfs/qmuy4jh5mgnzvlkjies1rwm4yuvjh5o2fyopnpvywrrvgv.

[17] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping authorities" honest or bust" with decentralized witness cosigning. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 526–545. Ieee, 2016.

[18] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework, https://polkadot.network/polkadotpaper.pdf.

[19] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus in the lens of blockchain. *CoRR*, abs/1803.05069, 2018.

[20] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 347–356, New York, NY, USA, 2019. ACM.