Fast, Censorship Resistant BFT Consensus for Blockchains

Daniel Lubarov Brendan Farmer

DRAFT
April 22, 2019

Abstract

Byzantine fault tolerant (BFT) consensus algorithms allow blockchains to confirm transactions much more quickly than longest-chain systems like Bitcoin. We present a new partially synchronous BFT algorithm which achieves lower latency than existing solutions. Our algorithm takes fewer communication steps to terminate in certain cases, and under normal conditions, it transitions from one step to another based on observed messages rather than waiting for timeouts.

In simulations, our algorithm was approximately 14% faster than the other ones we tested when configured with optimal timeout parameters. This gap widened when larger timeout parameters were used, since our algorithm rarely makes state transitions based on timeouts.

Contents

1 Introduction **3**
1.1 Our contribution 3
1.2 Roadmap 4

2 Basic algorithm **4**
2.1 Assumptions 4
2.2 Round model 4
2.3 Definitions 5
2.4 Algorithm 5
2.5 Proof of agreement 6
2.6 Proof of termination 7

3 Eager termination **8**
3.1 The case against asynchronous algorithms 8
3.2 A hybrid round model 9
3.3 Analysis sketch 9

4 Tolerating unknown latency **10**

5 Related work **11**
5.1 Steps to termination 11
5.2 Responsiveness 12
5.3 Censorship and other soft forks 13

6 Evaluation **14**

1 Introduction

If cryptocurrencies are to be used in everyday commerce, it is imperative that transactions be confirmed quickly. While Bitcoin is notoriously slow at confirming transactions,[1] projects like Tendermint [2,3] and Algorand [4] have achieved much lower latency by adapting BFT consensus algorithms to a blockchain setting.

Tendermint and Algorand both operate in rounds consisting of three communication steps: a leader proposal followed by two voting steps. Under good conditions, both algorithms will terminate after a single round, but faulty leaders or network asynchrony can cause a round to fail, in which case the cycle continues with a new round.

While Tendermint and Algorand achieve much better latency than longest-chain consensus schemes, we argue that they are suboptimal in a few ways. First, since their rounds always consist of three steps each, any failure which impedes the current round will delay termination by at least three steps. Second, they both involve some fixed timeouts which we argue are unnecessary. Finally, they enable a faction of nodes with 34% voting power to censor block proposals by simply ignoring them.

1.1 Our contribution

We present a simple algorithm which, under good conditions, terminates after three communication steps. In that respect it is similar to Tendermint and Algorand, but it behaves differently in the event of late proposals or propagation delays. In our algorithm, each proposal is followed by a variable number of voting steps, which gives it more flexibility to be accept proposals even after certain steps fail.

For example, suppose that following an initial proposal, the first voting step fails to propagate quickly due to network delays. In Tendermint or Algorand, no progress could be made until the next round, implying a minimum of six steps before termination. Our algorithm would extend the first round with one extra voting step, after which the algorithm can terminate without needing a second round.

Our algorithm is not fully asynchronous—we argue against strict asynchrony in Section 3.1—but we use timeouts only as a fallback. Under good conditions, our algorithm transitions from one state to another based on observed messages rather than timeouts.

Finally, our algorithm raises the bar for censorship attacks. If a faction of nodes with 34% voting power were to ignore a proposal they didn't like, it would stall

[1] The original Bitcoin whitepaper [1] analyzed the probability that a malicious miner could undo a transaction. For example, an attacker with 30% hash power would have a 4.17% chance of undoing a transaction after 10 confirmations.

the consensus process rather than causing the proposal to fail.

1.2 Roadmap

In Section 2, we introduce the most basic version of our algorithm, which relies on absolute clocks and assumes a known network delay. We then cover a few optional variations of the algorithm. Section 3 introduces a different model for communication steps, allowing the algorithm to transition between steps as soon as certain votes are observed rather than waiting for timeouts. Section 4 relaxes our partial synchrony assumption somewhat, allowing the algorithm to work with an unknown network delay.

In Section 5, we compare our algorithm's performance and censorship resistance to that of a few related algorithms. Finally, in Section 6, we will evaluate our algorithm's concrete performance.

2 Basic algorithm

2.1 Assumptions

We will assume that $n > 3f$; in other words, we assume that less than one third of nodes are faulty. This is optimal in the sense that a greater number of Byzantine faults would make it impossible for any algorithm to achieve consensus, as shown in [5].

We say that the network is synchronous, with a propagation time bounded by δ, if all message delivery times among correct nodes are upper bounded by δ. We say that a network is synchronous over a period $[t_1, t_2]$ if this condition holds during that period. In particular, if a correct node broadcasts a message at time $t \in [t_1, t_2]$, and $t + \delta \in [t_1, t_2]$, then all correct nodes will receive the message by $t + \delta$.

We assume that the network is partially synchronous, in the sense that for any duration Δ, there will eventually be a period $[t, t + \Delta]$ such that the network is synchronous over that period. We assume that the propagation time δ is known, but we will relax this assumption in Section 4 to handle unknown propagation times.

2.2 Round model

Our algorithm is carried out over one or more rounds. Each round has a leader; we assume that these leaders have already been determined in advance.

Each round consists of several communication steps. We say that a step is synchronous if and only if the network is synchronous for the entire duration of

the step. We will sometimes use the notation (r, s) to refer to step number s of round number r.

For now, we will assume that each node is equipped with a clock which gives a perfect measure of absolute time. In Section 3, we will relax this assumption with a round model based on relative timing rather than absolute timing.

2.3 Definitions

We define two types of votes. $\langle \texttt{PREPARE}, v, b, r, s \rangle$ signifies that some node v votes to prepare block b in (r, s). $\langle \texttt{COMMIT}, v, b, r, s \rangle$ signifies that v votes to commit b in (r, s). Note that in either case, b may be $\perp$, representing an empty block. Both messages are accompanied by signatures to ensure authenticity.

Next, we define the predicate prepared(b, r, s) to be true if and only if b has received at least $2f + 1$ votes of either kind in (r, s). We define committed(b, r, s) to be true if and only if b has received at least $2f + 1$ `COMMIT` votes in (r, s).

Further, we define prepared$_v(b, r, s)$ to be true if and only if v knows prepared(b, r, s) to be true based on the votes that v has observed. Similarly, we define committed$_v(b, r, s)$ to be true if and only if v knows committed(b, r, s) to be true.

2.4 Algorithm

Each round r proceeds as follows. In the first step ($s = 0$), the preselected leader broadcasts a block proposal. In each subsequent step ($s \geq 1$), each node v votes according to the following rules:

1. If prepared$_v(b, r, s')$ for some block b and step number s', then without loss of generality, let s', b be the largest such round number and the associated block.
   (a) If $s' = s - 1$, then v votes $\langle \texttt{COMMIT}, v, b, r, s \rangle$.
   (b) Otherwise, v votes $\langle \texttt{PREPARE}, v, b, r, s \rangle$.
2. Otherwise, let B be the set of valid blocks that v has received for the current round.
   (a) If B is a singleton set $\{b\}$, then v votes $\langle \texttt{PREPARE}, v, b, r, s \rangle$.
   (b) Otherwise, v votes $\langle \texttt{PREPARE}, v, \perp, r, s \rangle$.

A correct node v will end round r as soon as committed$_v(b, r, s)$ for any b, s. If $b = \perp$, then v enters $(r + 1, 0)$. Otherwise, v terminates with b as its output.

2.5 Proof of agreement

We will now show that the algorithm is safe, in the sense that all correct nodes will eventually output the same (non-empty) block.

Lemma 1. committed(b, r, s) *implies* prepared(b, r, s).

Proof. This follows immediately from our definitions. committed(b, r, s) entails that b received at least $2f + 1$ `COMMIT` votes. This also satisfies the definition of prepared(b, r, s), which counts votes of either kind. □

Lemma 2. *At most one block will become prepared at each* (r, s).

Proof. Assume the opposite: $\exists b, b', r, s$ such that prepared(b, r, s) and prepared(b', r, s). Then both b and b' received $2f + 1$ or more votes in r, s, for a total of $4f + 2$ or more votes. Since there are only $3f + 1$ nodes, at least $f + 1$ of them must have voted for both b and b', which contradicts our assumption that at most f nodes are faulty. □

Lemma 3. *If some block is committed, no other block may be prepared in a later step of the same round.*

Proof. Suppose committed(b, r, s). Then by definition, $2f + 1$ nodes must have voted $\langle \texttt{COMMIT}, v, b, r, s\rangle$, and by our correctness assumption, at least $f + 1$ of these votes must have come from correct nodes. For each such node v followed rule 1a, we know that prepared$_v(b, r, s - 1)$.

By the logic of rule 1, these $f + 1$ correct nodes will continue voting for b until a different block becomes prepared. This leaves $(3f + 1) - (f + 1) = 2f$ remaining nodes which may vote for a different block. Since $2f$ is one short of a quorum, a different block will never acquire enough votes to become prepared. □

Lemma 4. *At most one block may be committed in any given round.*

Proof. Suppose, to the contrary, that b was committed in (r, s) while a distinct block b' was committed in (r, s'). Lemma 2 implies $s \neq s'$. Without loss of generality, assume $s < s'$. Lemma 1 implies prepared(b', r, s'), which contradicts Lemma 3. □

Theorem 1 (Agreement)**.** *At most one non-empty block may be committed.*

Proof. Suppose, to the contrary, that committed(b, r, s) and committed(b', r', s') for distinct non-empty blocks $b \neq b'$. Lemma 4 implies that $r \neq r'$. Without loss of generality, assume $r < r'$. $\perp$ could not have been committed in round r, as that would violate Lemma 4. So correct nodes could not have progressed beyond round r, making it impossible for any block to be committed in round r'. □

2.6 Proof of termination

To facilitate our proof of termination, we will define a state machine for a particular round of the algorithm. Let (r, s) be the current state. Let B be the set of valid blocks which the leader has broadcasted. Let $B' \subseteq B$ be the subset of these blocks which have been received by at least $2f+1$ correct nodes before they entered (r, s).

Consider these possible round states:

1. $\nexists b, s'$ such that prepared(b, r, s'), and $|B'| = 0$.
2. $\nexists b, s'$ such that prepared(b, r, s'), and $|B'| = 1$.
3. $\nexists b, s'$ such that prepared(b, r, s'), and $|B'| \geq 2$.
4. $\exists b, s'$ such that prepared(b, r, s'), but all such $s' < s - 1$.
5. $\exists b$ such that prepared$(b, r, s - 1)$.
6. $\exists b, s'$ such that committed(b, r, s').

Note that the last state, 6, is terminal in that it signifies the end of round r.

Lemma 5. *If round r has not yet terminated, then after a synchronous step, r will necessarily progress to a higher-numbered state.*

Proof. If the current state is 1 and $|B| = 0$, then no block proposals will be observed in this step, so all correct nodes will vote to prepare $\perp$ in accordance with rule 2b, advancing us to state 5. On the other hand, if $|B| > 0$, then all blocks in B will propagate during this step, advancing us to either state 2 (if $|B| = 1$) or state 3 (if $|B| \geq 2$).

If the current state is 2, the logic is similar. If $|B| = 1$, then all correct nodes will vote to prepare the unique block $b \in B$, advancing us to state 5. If $|B| \geq 2$, then block propagation will advance us to state 3.

If the current state is 3, then all correct nodes will vote to prepare $\perp$ based on rule 2b, advancing us to state 5.

If the current state is 4, then all correct nodes will vote to prepare the block b which was prepared most recently, in accordance with rule 1b. Consequently b will become prepared this round, advancing us to state 5.

If the current state is 5, then all correct nodes will vote $\langle \texttt{COMMIT}, v, a, r \rangle$ in accordance with rule 1a, thereby advancing us to state 6 which is terminal. □

Lemma 6 (Round termination)**.** *Every round eventually terminates.*

Proof. Let $\Delta = 6\delta$. Our partial synchrony assumption entails there there will eventually come a period $[t, t+\Delta]$ during which the network is synchronous. There will be at least five synchronous steps during that period (six if t happens to align with a step boundary). Since there are only six states, Lemma 5 implies that we will necessarily end up in the terminal sixth state after these five synchronous steps. □

Theorem 2 (Termination)**.** *Every correct node eventually outputs some non-empty block.*

Proof. Our proof follows Lemma 6, but this time with $\Delta = 9\delta$. By our partial synchrony assumption, there will eventually come a period $[t, t + \Delta]$ with eight synchronous steps. Let r be the round number at time t; then Lemma 6 implies that r will terminate within the first five of these synchronous steps. This leaves at least three synchronous steps for round $r + 1$.

Without loss of generality, assume that the leader of round $r + 1$ is correct. Our correctness assumption implies that this condition will almost surely hold eventually.

Round $r+1$ begins in state 1, and the correctness of the leader implies $B = \{b\}$ for some non-empty block b. In step 0, all correct nodes will observe b, advancing us to state 2. In step 1, each correct node v will vote $\langle \texttt{PREPARE}, v, b, r + 1, 1\rangle$ in accordance with rule 2a, advancing us to state 5. In the third step, each correct node v will vote $\langle \texttt{COMMIT}, v, b, r+1, 2\rangle$ in accordance with rule 1a. Since $b \neq \perp$, each correct node v will terminate as soon as they observe $\text{committed}_v(b, r + 1, 2)$. □

3 Eager termination

In Section 2.2, we used a simple model where each communication step takes place over a period of δ (the propagation time). The timing of rounds was based on absolute clocks, which could be problematic in cases where certain nodes do not have precisely synchronized clocks. Moreover, since propagation delays vary based on network conditions, it seems suboptimal to require that every communication step take the same amount of time.

3.1 The case against asynchronous algorithms

Asynchronous algorithms offer a solution to this problem, since they make state transitions exclusively based on observed messages rather than using timers. However, it is impossible for a strictly asynchronous and deterministic consensus algorithm to guarantee termination in the presence of Byzantine faults [6]. Asynchronous algorithms avoid this impossibility result by sacrificing determinism, but doing so negatively affects average termination times.

For example, the binary consensus algorithm used by Honey Badger [7] has a 50% chance of termination in each round, so the number of rounds before termination, R, is geometrically distributed with a trial probability of 0.5. The algorithm will almost surely terminate given infinite time, since $\lim_{r\to\infty} F_R(r) = 1$ by the definition of a CDF. Since $E(R) = 2$ however, the algorithm will take two rounds on average. And since each round involves several communication steps, the algorithm is not a good fit for latency critical applications such as real time payments.

These concerns lead us to introduce a new round model, which can be viewed as a compromise between fixed-length communication steps and strictly asynchronous models.

3.2 A hybrid round model

In the basic version of our algorithm, there is no voting during the proposal step. While it is not essential, having voting in every step allows us to use the same state transition mechanism for each step of the algorithm. Thus we introduce a new type of vote, $\langle \texttt{BEGIN\_PROPOSE}, v, r\rangle$, which indicates that a node v has begun the proposal step of round r.

In our model, a correct node v enters state (r, s) if they have never entered (r, s), they have never entered $(r + 1, 0)$, and one of the following conditions is detected:

1. $s = 0$ and $\text{committed}_v(\perp, r - 1, s')$ for some s'.
2. $s = 1$ and v has observed a proposal for round r.
3. $s \geq 2$ and $\text{prepared}_v(b, r, s - 1)$ for some b.
4. $s \geq 1$ and a timeout of 2δ has elapsed after $f + 1$ votes were observed in $(r, s - 1)$.

3.3 Analysis sketch

(TODO: This section is unfinished.)

4 Tolerating unknown latency

So far, we have assumed a known upper bound δ on propagation times. In packet switched networks, however, propagation times can vary significantly due to congestion. Any proposed value for δ would be "too small" during periods of high congestion, and simultaneously "too large" during periods of low congestion.

One way to avoid this dilemma is to increase δ automatically after the algorithm fails to make progress. This idea dates back to DLS [8], and similar mechanisms have been suggested for PBFT [9] and Tendermint [2]. Our algorithm requires a slightly different solution, however, due to its unique round structure.

In algorithms with rounds consisting of a fixed number of steps, such as Tendermint, timeouts can simply be increased after each unsuccessful round. If we adopted the same approach, however, the first round of our algorithm might never end. In order to maintain liveness, then, we must increase timeouts after a certain number of steps within a round.

Since each round of our algorithm takes three steps under good conditions, we propose adjusting the timeout after three steps if no proposal has been committed yet. We also propose adjusting the timeout every two steps thereafter, since it takes two synchronous steps of voting to commit a proposal. Let $a(s)$ be the number of adjustments applied to a given step number s; then our proposal is

$$a(s) = \begin{cases} \lfloor (s-1)/2 \rfloor & \text{if } s \geq 3 \\ 0 & \text{otherwise} \end{cases}$$

Note that if $\perp$ is committed in step s of round r, the following round $r+1$ should begin with $a(s)$ initial adjustments. If we were to revert back to the initial timeout δ_0 after every round, it is possible that $\perp$ would be committed repeatedly, if δ_0 did not allow enough time for proposals to propagate.

δ_s should increase monotonically with $a(s)$. The exact relationship is left as an implementation detail, but we suggest an exponential function, such as

$$\delta_s = \delta_0 \cdot 2^{a(s)}$$

as in [9]. This ensures that the algorithm's timeout converges with the actual propagation delay of the network, δ_{net}, after $\mathcal{O}(\log(\delta_{\text{net}}/\delta_0))$ adjustments. If δ_s were linear in $a(s)$, convergence would instead take $\mathcal{O}(\delta_{\text{net}}/\delta_0)$ adjustments.

5 Related work

As we mentioned in Section 1, our algorithm most closely resembles Tendermint [3] and Algorand [4]. Here we will give a more detailed comparison between the three algorithms.

5.1 Steps to termination

To start, we assume that all nodes are correct and the network is synchronous. In this setting, each algorithm takes three steps to terminate, as shown:

Step	Network	Tendermint	Algorand	This Work
1	Sync	First leader proposes b		
2	Sync	All pre-vote b	All soft-vote b	All prepare-vote b
2	Sync	All pre-commit b (terminal)	All cert-vote b (terminal)	All commit-vote b (terminal)

Next, consider the case where all nodes are correct, but the first leader's proposal is not propagated quickly due to a network delay. In particular, suppose that 50% of nodes receive the first proposal during the first communication step, and 50% do not. As we can see, our algorithm outperforms the others in this case:

Step	Network	Tendermint	Algorand	This Work
1	Async	First leader proposes b_1		
2	Sync	50% pre-vote b_1; 50% pre-vote $\perp$	50% soft-vote b_1; 50% soft-vote $\perp$	50% prepare-vote b_1; 50% prepare-vote $\perp$
3	Sync	All pre-commit $\perp$	No cert-votes	All prepare-vote b_1
4	Sync	Second leader re-proposes b_1	All next-vote $\perp$	All commit-vote b_1 (terminal)
5	Sync	All pre-vote b_1	Second leader proposes b_2	
6	Sync	All pre-commit b_1 (terminal)	All soft-vote b_2	
7	Sync		All cert-vote b_2 (terminal)	

There is also a case in which our algorithm underperforms the others. Suppose that the network is synchronous but the first leader is malicious, and wishes to delay consensus for as long as possible. They could do so by broadcasting proposals mid-way through the proposal step, so that when nodes vote in the following step, some will have observed the proposal while others will not have. This can prevent any value from becoming prepared for a while, although the damage is limited because once all correct nodes have observed two or more proposals, they will all vote to prepare $\perp$ based on rule 2b.

Tendermint and Algorand would perform exactly as above, taking six and seven steps respectively, but our algorithm would take eight steps to commit a non-empty block in this scenario:

Step	Network	This Work
1	Sync	First leader proposes b_1 mid-way through the round
2	Sync	50% prepare-vote b_1; 50% prepare-vote $\perp$ First leader proposes b_2 mid-way through the round
3	Sync	50% prepare-vote b_1; 50% prepare-vote $\perp$
4	Sync	All pre-vote $\perp$
5	Sync	All commit-vote $\perp$
6	Sync	Second leader proposes b_3
7	Sync	All pre-vote b_3
8	Sync	All commit-vote b_3

On average, we expect that our algorithm will terminate in fewer steps than Tendermint or Algorand as long as the frequency of network delays exceeds the frequency of malicious leaders who wish to delay consensus.

5.2 Responsiveness

Jae Kwon's original Tendermint concept [2] used a fixed round model, where "the Propose, Prevote, and Precommit steps each take one third of the total time allocated for that round." Kwon did suggest increasing the duration of each round after unsuccessful rounds, similar to the mechanism we described in Section 4. Still, the algorithm is not fully responsive in the sense that it makes state transitions based on timeouts rather than observed messages.

Ethan Buchman's thesis [3] improved upon the original Tendermint concept by terminating both the pre-vote and pre-commit steps after 2/3 votes had been received. The trouble with this approach is that if $\perp$ appears among the first 2/3 votes observed by a node, the node will terminate the vote before observing a quorum. A malicious node could vote for $\perp$ repeatedly in order to stall consensus. Buchman recognized this issue, and suggested that "we can sleep for some amount of time to give slower or delayed votes a chance to be received." Still, to the best of our knowledge, there has not been a concrete proposal for a responsive variant of Tendermint.

Algorand can be viewed as partially responsive, since it makes certain state transitions based on observed messages. For instance, there are no timeouts during Algorand's second finishing step; the only way to exit that state is by observing a certain number of next-votes. On the other hand, other states can only be exited via timeouts, such as the proposal step.

By contrast, our protocol uses timeouts only as a fallback. During periods of synchrony, all voting steps will be terminated based on observed votes. And in rounds with a correct leader, the proposal step is terminated by observing the leader's proposal. In Section 6, we will show that this added responsiveness can significantly improve average termination times.

5.3 Censorship and other soft forks

A soft fork is a fork of a blockchain in which the fork's rules are stricter than those of the parent chain. If a soft fork gains enough support, it will be accepted by nodes which enforce the parent chain's rules, since any block in a soft fork is also valid under the rules of its parent. In Bitcoin, for example, a soft fork with 51% hash power will eventually be accepted by all nodes, whether they are enforcing the soft fork's rules or the original rules.

Soft forks can be used to deploy upgrades without breaking backwards compatibility. An example of such is Bitcoin's Segregated Witness upgrade, where a soft fork was used to prohibit transactions with malleable fields. On the other hand, soft forks could potentially be used to censor transactions from certain blacklisted entities. Even in a protocol like ZeroCash [10], where such censorship would be impractical, soft forks could be used for other malicious purposes. For example, a large coalition of miners could use a soft fork to censor blocks from any miners outside the coalition.

In longest-chain protocols such as Bitcoin, a soft fork requires 51% hash power in order to be accepted by miners who enforce the rules of the parent chain. In BFT consensus systems, the requirements are a bit different. Suppose that a coalition of $f+1$ nodes wished to perform a soft fork, while the remaining $2f$ nodes behaved correctly according to the original rules. Suppose that in round r, a block b was proposed which the coalition wished to censor.

In the context of Tendermint or Algorand, the coalition could simply ignore b. The $2f$ correct nodes would be one vote short of obtaining a quorum for b, so the result of round r would be $\perp$. If a correct node was the leader of round $r+1$, it might propose b a second time, but the coalition could repeatedly force $\perp$ to be committed. Eventually a coalition member would become the leader, at which point they would be free to propose a different block.

In the context of our algorithm, this strategy would play out differently, since round r would not end until either b or $\perp$ obtained a quorum. The coalition could again prevent b from obtaining a quorum by simply ignoring it, but correct nodes would repeatedly vote to prepare b. The coalition could stall the protocol by abstaining, but they would have no way of getting $\perp$ or a different block committed.

6 Evaluation

We implemented Tendermint, Algorand and our own algorithm with a simulated network. The simulated network contained 100 nodes, 90 of which were available, and 10 of which were offline. We did not simulate malicious nodes, since each algorithm is susceptible to different kinds of malicious behavior, making a comparison difficult.

We assigned each node to a random location on the Earth. For each pair of nodes, we calculated a "best case" latency with the assumption that the nodes were directly connected by a fiber optic cable. In particular, we took the great-circle distance between the nodes' locations and divided it by c/n, where c is the speed of light, and $n = 1.4682$ is the refractive index of a typical fiber optic cable. For each message, we scaled this best case latency by a random multiplier in $[1, 2]$ to simulate network delays.

We then invoked each algorithm with various timeout configurations. To mitigate random error, we ran each algorithm 1000 times per configuration and took the average latency.

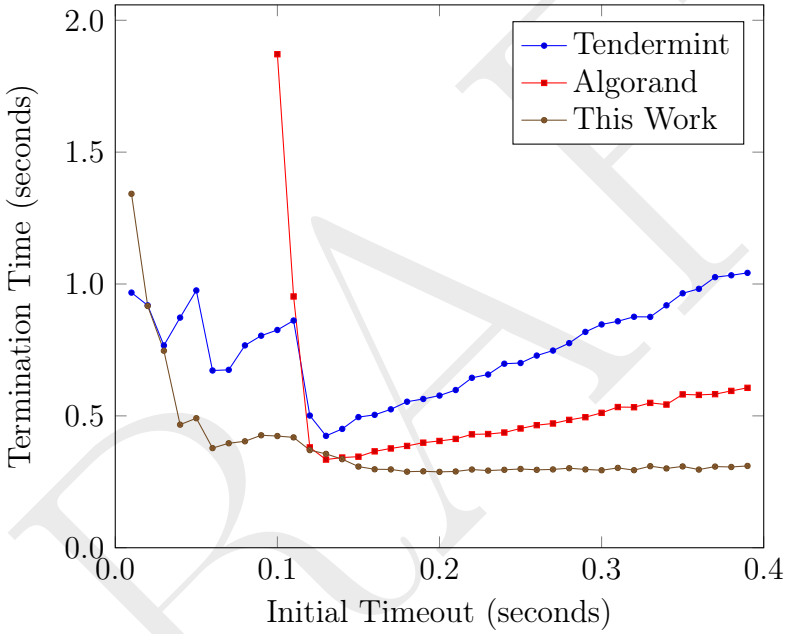

Figure 1: The time at which each algorithm terminated in the simulation, given a particular initial timeout.

The simulation results are shown in Figure 1. Note that there is no data for Algorand with very small timeouts, because the algorithm did not terminate in those cases. Algorand could be adapted to use increasing timeouts, but our implementation closely follows the Algorand paper, which does not specify such a mechanism.

As the data shows, the performance of Tendermint and Algorand depends heavily on the initial timeout parameter, since both algorithms make certain state transitions based on timeouts. Our algorithm achieves significantly better latency when large initial timeouts are used, since it normally makes state transitions based on observed messages before any timeouts are triggered.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] J. Kwon, "Tendermint: Consensus without mining," *Retrieved May*, vol. 18, p. 2017, 2014.

[3] E. Buchman, *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[4] J. Chen, S. Gorbunov, S. Micali, and G. Vlachos, "Algorand agreement: Super fast and partition resilient byzantine agreement." Cryptology ePrint Archive, Report 2018/377, 2018. https://eprint.iacr.org/2018/377.

[5] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 824–840, 1985.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process.," tech. rep., Massachusetts Inst of Tech Cambridge Lab for Computer Science, 1982.

[7] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 31–42, ACM, 2016.

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

[9] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, pp. 173–186, 1999.

[10] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 459–474, IEEE, 2014.