How robust are synchronous consensus protocols?

Nenad Milošević
Università della Svizzera italiana (USI), Lugano, Switzerland

Daniel Cason
Informal Systems, Toronto, Canada

Zarko Milošević
Informal Systems, Toronto, Canada

Fernando Pedone
Università della Svizzera italiana (USI), Lugano, Switzerland

Abstract

Synchronous Byzantine fault-tolerant (BFT) protocols have long been a reality in an academic setting, yet their practicality remains debated. The main concern of skeptics of synchronous systems is that the correctness of these protocols depends on the timely delivery of all messages within a predefined synchronous bound, Δ. This dependency creates a challenging tradeoff between protocol correctness and performance, as Δ directly impacts both. In this paper, we examine this tradeoff in detail. Specifically, we introduce BoundBFT, a new synchronous BFT consensus protocol. We analyze how BoundBFT's correctness can be compromised and use this analysis to design and implement the most effective attack strategies that malicious processes could employ. Furthermore, we experimentally determine the synchronous bound Δ that provides sufficient confidence in maintaining protocol correctness even in the presence of malicious replicas. Finally, we apply this discovered bound to BoundBFT, evaluate its performance, and compare it to state-of-the-art synchronous and partially synchronous protocols.

2012 ACM Subject Classification Computer systems organization → Reliability; Computer systems organization → Availability; Computer systems organization → Redundancy; Computing methodologies → Distributed algorithms

Keywords and phrases Synchronous Consensus, Byzantine Failures, Blockchain

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2024.28

Funding This work was partially supported by the Swiss National Science Foundation (grant 175717).

1 Introduction

Synchronous consensus protocols have long been a topic of debate in robust distributed systems. On the one hand, synchronous consensus protocols can tolerate $f < n/2$ Byzantine or malicious processes out of n processes [17, 15, 26], an improvement over partially synchronous consensus protocols, which require $f < n/3$ [13]. On the other hand, the correctness of a synchronous protocol hinges on the timely delivery of messages within a fixed time bound, Δ. To ensure synchrony is not violated (i.e., messages are delivered within Δ), existing synchronous consensus protocols set Δ conservatively, as the 99.99-th percentile of sampled communication [30] or as a 10-time factor of average latency [2]. This reliance on Δ presents a critical tradeoff: a more conservative Δ minimizes the risk of synchrony violations, thus favoring correctness, but comes at the cost of reduced protocol performance, since synchronous protocols execute at the pace of Δ.

This paper offers a new perspective on synchronous systems by starting with the observation that some synchronous consensus protocols are resilient to synchrony violations; that is, even if some messages are not delivered within Δ, correctness is not compromised. As we now illustrate, resilience to synchrony violations happens due to communication diversity

and redundancy in a protocol. In Figure 1 (left), process p sends request m_A to process q and sets a 2Δ timeout for the answer from q. Even if p's request violates synchrony (i.e., m_A takes longer than Δ to arrive at q), q's response (m_B) makes up for the delay and arrives at p within the expected 2Δ. In Figure 1 (right), p sends request m_A to q and r and sets a 3Δ timeout for their answer. Process q receives m_A timely, replies to p (m_B) and relays m_A to r. Although r receives m_A from p after Δ, it receives m_A from q timely and responds to p (m_C). As a result, p receives responses from q and r within the expected 3Δ. These communication patterns are at the core of BoundBFT, a novel Byzantine fault-tolerant synchronous consensus protocol introduced in the paper.

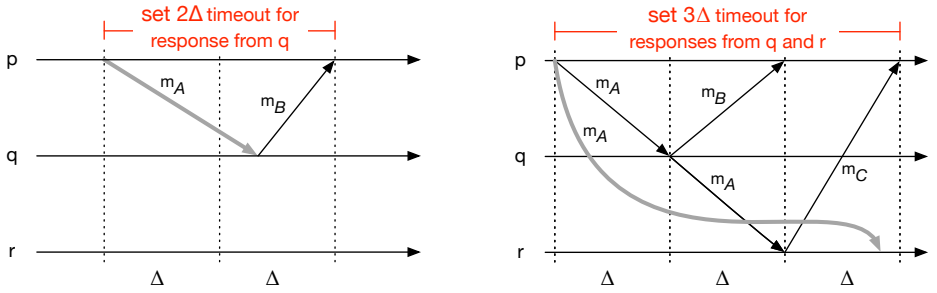

Figure 1 BoundBFT execution patterns where gray messages violate synchronous bound Δ but do not compromise protocol correctness.

Tolerating synchrony violations can provide substantial improvement in performance. To understand why, consider Table 1, reproduced from [30], which compares the 99.99-th and 99.999-th percentile of communication across Amazon EC2 datacenters. While a protocol that can tolerate one synchrony violation in every one hundred thousand messages exchanged between US West and US East must set Δ to at least 82190 milliseconds, a protocol that tolerates a synchrony violation in every ten thousand transmitted messages can set Δ to 1097 milliseconds. Since synchronous consensus protocols run at the pace of Δ, this represents a $75\times$ improvement in performance!

Since BoundBFT tolerates Byzantine failures, synchrony violations should not introduce vulnerabilities that could be exploited by malicious processes. In leader-based consensus protocols that tolerate Byzantine failures, such as BoundBFT, the leader is the most advantageous role for a malicious process as it can induce honest processes into inconsistent decisions, possibly with help from other malicious processes. In a synchronous protocol, the malicious leader can hope to get "additional help" from synchrony violations, as some honest processes may be delayed with respect to other processes. The attack will work as long as the deceived honest process does not find out about the trickery before deciding. But honest processes communicate with many other honest processes, so there is ample opportunity to find out about the attack even if some messages are delayed.

In this paper, we assess the robustness of BoundBFT, that is, its ability to maintain correctness under synchrony violations, both in the presence and absence of malicious processes. Leveraging BoundBFT's leader-based execution model with signed messages, we first characterize the range of potential attacks and examine their effects when combined with synchrony violations. We then design and implement specific Byzantine attacks to rigorously evaluate BoundBFT's robustness. Namely, we conduct experiments to determine an appropriate synchrony bound, Δ, that provides high confidence in preserving protocol correctness, even under the attack. Finally, we apply this bound to evaluate BoundBFT's performance, enabling meaningful comparison with partially synchronous protocols.

	US West (CA)		Europe (EU)		Tokyo (JP)		Sydney (AU)		Sao Paolo (BR)	
	99.99%	99.999%	99.99%	99.999%	99.99%	99.999%	99.99%	99.999%	99.99%	99.999%
US East (VA)	1097	82190	1112	85649	1226	81177	1372	95074	1214	85434
US West (CA)			1184	1974	1133	1180	1209	6354	1252	90980
Europe (EU)					1310	1397	1375	3154	1257	1382
Tokyo (JP)							1149	1414	2496	11399
Sydney (AU)									1496	2134

Table 1 Round-trip latency (in milliseconds) of hping3 across Amazon EC2 datacenters, collected during three months [30].

We have implemented BoundBFT and compared it to state-of-the-art synchronous (Sync HotStuff [2, 3]) and partially synchronous consensus protocols (Tendermint [5] and HotStuff-2 [32]). Our evaluation in an emulated geographically distributed system showed that BoundBFT's synchrony bounds can be in some cases more than one order of magnitude smaller than usual conservative synchronous bounds [30]. As a result, BoundBFT achieves latency comparable to partially synchronous consensus protocols, while offering higher throughput, reliability, and availability.

The remainder of the paper is structured as follows. Section 2 defines the system model and introduces background information on blockchain. Section 3 presents BoundBFT, a new Byzantine fault-tolerant consensus algorithm designed for the synchronous system model. Section 4 analyzes BoundBFT under synchrony violations and attacks. Section 5 experimentally evaluates BoundBFT and competing approaches. Section 6 overviews related work and Section 7 concludes. The Appendix contains BoundBFT's proof of correctness and the full data of our experimental evaluation.

2 Background

2.1 System model

We consider a message-passing geographically distributed system consisting of a set of processes (or replicas) that do not have access to a shared memory or a global clock. Each process has its own local (hardware) clock, and while these clocks are not synchronized, they all run at the same speed. Processes can be *honest* or *faulty*. An honest process follows its specification; a faulty or Byzantine process presents arbitrary behavior. There are f faulty processes out of n processes. Processes communicate using point-to-point reliable links: every message an honest sender sends to an honest receiver is received.

We assume a *synchronous* system: there exists a known bound Δ on maximal network transmission delay in communication between honest processes. We do not assume lock-step execution (e.g., [28, 12]); instead, we assume that all honest replicas start the execution within Δ time. We compare our proposed synchronous protocol to protocols that assume *partial synchrony*: the system is initially asynchronous, without communication bounds, and eventually becomes synchronous.

We use cryptographic techniques for authentication, and digest calculation. We assume that adversaries (and Byzantine processes under their control) are computationally bound so that they are unable to subvert the cryptographic techniques used. Adversaries can coordinate Byzantine processes but cannot delay honest processes.

2.2 Blockchain

A blockchain is a distributed append-only log of transactions implemented by geographically distributed processes. A blockchain (consensus) protocol forms a chain of blocks, where a

block's position in the chain is the block's *height*. A block B_k at height k has the following format $B_k \coloneqq (b_k, H(B_{k-1}))$ where b_k denotes a proposed value (i.e., a set of transactions) and $H(B_{k-1})$ is a hash digest of the predecessor block. The first block, $B_1 = (b_1, \perp)$, has no predecessor. Every subsequent block B_k must specify a predecessor block B_{k-1} by including a hash of it. A block is valid if (i) its predecessor is valid or $\perp$, and (ii) its proposed value meets application-level validity conditions and is consistent with its chain of ancestors (e.g., there are no double-spending transactions). If block B_k is an ancestor of block B_l (i.e., $l \geq k$), we say B_l *extends* B_k. We say blocks B_l and B'_l, *equivocate* each other if they do not extend one another.

We assume that a blockchain (consensus) protocol must satisfy the following properties:

- *Agreement*: No two honest replicas commit different blocks at the same height.
- *Progress*: All honest replicas keep committing new blocks.
- *External validity*: Every committed block satisfies the predefined *valid()* predicate.

3 BoundBFT

3.1 The protocol

BoundBFT is a synchronous BFT consensus protocol with a rotating leader [3]. It tolerates up to $f < n/2$ Byzantine replicas. BoundBFT adopts the good-case execution of the rotating-leader version of Sync HotStuff [3, 2], achieving optimal latency and responsive leader rotations [3]. However, it introduces a different epoch synchronization mechanism that reduces the waiting time for a new leader to propose from 9Δ to 5Δ when the previous leader is silent.

Algorithm 1 presents BoundBFT's pseudo-code that covers executions when leaders are honest. BoundBFT's execution evolves as a sequence of epochs, numbered $0, 1, 2, \ldots$, with each replica tracking the last epoch it started, denoted as e_p. Each epoch e has a designated leader, computed using a deterministic function $leader(e)$.

At the start of an epoch, the leader l broadcasts the proposal containing a new block b that extends the most recently certified block it knows of, $validBlock_l$ (lines 14–18 in Algorithm 1). Along with the new block, the leader includes the certificate for $validBlock_l$, $validBC_l$.

Upon receiving a proposal (lines 19–25 in Algorithm 1), a replica verifies the proposal's validity and votes for it if the leader's block certificate is at least as recent as the replica's $lockedBC_p$. The replica votes by sending a signed vote message to all replicas. A vote contains the current epoch number and the hash of the block, $id(b)$.

When a replica receives a proposal and $f + 1$ votes for it, it forms a block certificate for the proposed block. If the replica has no proof of leader l misbehaving, it locks on b and triggers $timeoutCommit(e, b)$ (lines 26–32 in Algorithm 1). The replica then updates its $validBlock_p$ and $validBC_p$ variables (lines 33–36 in Algorithm 1) and starts epoch $e + 1$. To ensure all honest replicas receive the proposal and its certificate, the replica forwards them (lines 25 and 35 in Algorithm 1), allowing all honest replicas to start epoch $e + 1$ within Δ time.

When $timeoutCommit(e, b)$ expires and the replica has no evidence of leader misbehavior for epoch e, it commits block b and all blocks b extends (lines 37–40 in Algorithm 1). In other words, it directly commits block b and indirectly commits all its uncommitted ancestor blocks.

The replica does not wait for $timeoutCommit(e, b)$ to expire before starting the next epoch. Instead, it begins epoch $e + 1$ immediately after receiving a block certificate in epoch

Algorithm 1 BoundBFT consensus algorithm: normal case

```
1: Initialization:
2:   e_p := 0                                                          ▷ the current epoch
3:   hasVoted_p := false                                               ▷ has the replica voted in the current epoch?
4:   validBC_p := nil                                                  ▷ the most recent block certificate the replica is aware of and...
5:   validBlock_p := nil                                               ▷ the block certified by validBC_p
6:   lockedBC_p := nil                                                 ▷ the block certificate the replica is locked on and...
7:   lockedBlock_p := nil                                              ▷ the block certified by lockedBC_p
8:   epochsState_p[] := nil          ▷ an epoch can be in one of the states: ACTIVE, COMMITTED, NOT-COMMITTED
9: when bootstrapping do StartEpoch(0)                                 ▷ the execution starts in epoch 0
10: Procedure StartEpoch(e) :                                          ▷ upon starting an epoch:
11:   e_p ← e                                                          ▷ the replica resets the current epoch variables
12:   epochsState_p[e_p] ← ACTIVE
13:   hasVoted_p ← false
14:   if leader(e_p) = p then                                          ▷ if the replica is the leader in the current epoch...
15:     block.txs ← GetTxs()                                          ▷ it gets new transactions to include in the new block
16:     if validBlock_p ≠ nil then                                     ▷ then, if it knows of a previously certified block...
17:       block.prev ← id(validBlock_p)                                ▷ it links the new block with that block
18:     broadcast ⟨PROPOSE, e_p, block, validBC_p⟩_p                   ▷ lastly, it broadcasts the proposal with the new block...
                                                                       ▷ and the certificate for the block it is extending, validBC_p
19: when receive ⟨PROPOSE, e, b, BC⟩_l where valid(b) and              ▷ upon receiving the valid proposal...
20:  l = leader(e) and e = e_p do                                      ▷ from the leader of the current epoch:
21:   if epochsState_p[e] = ACTIVE ∧ hasVoted_p = false ∧              ▷ if the epoch is still active, replica has not voted yet, and...
22:    BC.epoch ≥ lockedBC_p.epoch then                                ▷ proposal's BC is at least as recent as replica's lockedBC...
23:     broadcast ⟨VOTE, e_p, id(b)⟩_p                                 ▷ the replica votes for a proposal, VOTE message contains block's hash
24:     hasVoted_p = true                                              ▷ then, the replica sets hasVoted_p so it does not vote twice, and...
25:     forward ⟨PROPOSE, e, b, BC⟩_l                                  ▷ forwards the proposal message
26: when receive ⟨PROPOSE, e, b, BC⟩_l and f + 1 distinct ⟨VOTE, e, id(b)⟩_* ▷ when the replica receives a proposal and...
27:  where e = e_p do                                                  ▷ f + 1 votes from the current epoch:
28:   cert ← NewCert from f + 1 ⟨VOTE, e, id(b)⟩_*                     ▷ it forms a block certificate
29:   if epochsState[e] = ACTIVE then                                  ▷ if no misbehavior is noticed in the current epoch...
30:     lockedBC_p ← cert                                              ▷ the replica locks on this block by setting lockedBC_p to cert and...
31:     lockedBlock_p ← b                                              ▷ lockedBlock_p to b, and...
32:     start timeoutCommit(e_p, b)                                    ▷ starts timeoutCommit
33:   validBC_p ← cert                                                 ▷ the replica always updates its validBC_p and validBlock_p...
34:   validBlock_p ← b                                                 ▷ to the most recent block
35:   forward messages from cert                                       ▷ lastly, the replica forwards the votes to other replicas and...
36:   StartEpoch(e + 1)                                                ▷ starts the next epoch
37: when timeoutCommit(e, b) expires do                                ▷ when timeoutCommit expires and...
38:   if epochsState[e] = ACTIVE then                                  ▷ the replica did not observe any proof of misbehavior,
39:     epochsState[e] ← COMMITTED                                     ▷ the replica commits the block b and...
40:     CommitBlockAndItsAncestors(b)                                  ▷ all its already uncommited ancestor blocks
```

e (line 36 in Algorithm 1). This approach allows BoundBFT to change leaders without waiting for the conservative network delay Δ when we have a sequence of honest leaders, a property known as *optimistic responsiveness* [32]. Additionally, BoundBFT implements *pipelining* [39], enabling replicas to start working on the next block before committing the previous one. Specifically, the leader in epoch $e + 1$ will propose a new block once it receives a block certificate for a block in epoch e.

Algorithm 2 presents BoundBFT's pseudo-code responsible for handling Byzantine leaders. To detect a malicious leader, a replica r starts a timer, $timeoutCertificate(e)$, when it enters epoch e (line 2 in Algorithm 2). If $timeoutCertificate(e)$ expires and r is still in epoch e, it indicates that r did not receive a block certificate, which can only occur if the leader is Byzantine. Consequently, replica r blames the leader and broadcasts a message $\langle \text{BLAME}, e \rangle_r$ (lines 3–5 in Algorithm 2).

When a replica receives $f + 1$ blame messages for epoch e from distinct replicas, it has proof that at least one honest replica blamed the leader and forms a blame certificate $C_e(\text{BLAME})$ (lines 6–7 in Algorithm 2).

Additionally, if an honest replica receives proposals for two distinct blocks signed by the leader in the same epoch e, it has proof that the leader is misbehaving. The replica then

Algorithm 2 BoundBFT consensus algorithm: handling malicious leaders

```
1:  upon starting the epoch e do                                   ▷ when a replica enters a new epoch:
2:      start timeoutCertificate(e_p)                              ▷ it starts the timer used to detect a malicious leader
3:  when timeoutCertificate(e) expires do                          ▷ when timeoutCertificate expires...
4:      if e = e_p ∧ epochsState[e] = ACTIVE then                  ▷ in the current epoch that is still ACTIVE
5:          broadcast ⟨BLAME, e_p⟩_p                               ▷ the replica blames the leader by broadcasting a BLAME message
6:  when receive f + 1 distinct ⟨BLAME, e⟩_* do                     ▷ when receiving f + 1 distinct BLAME messages from an epoch:
7:      cert ← NewCert from f + 1⟨BLAME, e⟩_*                       ▷ the replica forms a blame certificate and...
8:      MissbehaviorDetected(cert, e)                              ▷ calls MissbehaviorDetected with the certificate and epoch as parameters
9:  when receive ⟨PROPOSE, e, b, BC⟩_p and ⟨PROPOSE, e, b', BC'⟩_p  ▷ when replica receives two proposals...
10:     where p = leader(e) and b ≠ b' do                          ▷ from leader for two distinct blocks:
11:     cert ← NewCert from ⟨PROPOSE, e, b, BC⟩_p and ⟨PROPOSE, e, b', BC'⟩_p   ▷ the replica forms...
12:     MissbehaviorDetected(cert, e)                              ▷ an equivocation certificate and calls MissbehaviorDetected
13: Procedure MissbehaviorDetected(cert, e) :                      ▷ when misbehavior is detected in an epoch:
14:     if epochsState[e] = ACTIVE then                            ▷ if the epoch is still active
15:         epochsState[e] ← NOT-COMMITTED                         ▷ the replica sets state to NOT-COMMITTED
16:         if e = e_p then                                        ▷ if cert is the first certificate for the current epoch,
17:             forward messages from cert                         ▷ the replica forwards the messages from certificate and...
18:             start timeoutEpochChange(e_p)                      ▷ triggers timeoutEpochChange
19: when timeoutEpochChange(e) expires do                          ▷ when timeoutEpochChange expires:
20:     if e = e_p then                                            ▷ if the replica is in epoch e
21:         StartEpoch(e_p + 1)                                    ▷ the replica starts the next epoch
```

constructs an equivocation certificate $C_e(\text{EQUIV})$ (lines 9–11 in Algorithm 2).

Whenever a replica has proof of the leader's misbehavior (i.e., a blame or equivocation certificate), it calls the function $MissbehaviorDetected(cert, e)$ and forwards the certificate and epoch number to it (lines 8 and 12 in Algorithm 2). If a block is not committed in epoch e, the replica marks the epoch state as NOT-COMMITTED (line 15 in Algorithm 2). Moreover, if *cert* is the first certificate in epoch e, the replica forwards the certificate and triggers $timeoutEpochChange(e)$ (lines 16–18 in Algorithm 2). Forwarding the certificate ensures that all honest replicas learn that the leader is Byzantine within Δ time. Additionally, the extra timeout allows the replica to learn if an honest replica r moved to the next epoch before detecting leader misbehavior, i.e., r received a block certificate in epoch e, locked on it, and moved to the next epoch.

When $timeoutEpochChange(e)$ expires and the replica is still in epoch e, it moves to epoch $e + 1$ (lines 19–21 in Algorithm 2). Replicas wait for $timeoutEpochChange$ before moving to the next epoch only in the case of a Byzantine leader. If the leader is honest, replicas form a block certificate and move to the next epoch without waiting for any timeouts.

3.2 BoundBFT's correctness

In this section, we provide the intuition behind BoundBFT's correctness. The appendix contains a detailed correctness proof of BoundBFT.

3.2.1 Intuition behind epoch synchronization

The epoch synchronization mechanism guarantees that honest replicas progress through each epoch in a coordinated manner. Specifically, all honest replicas initiate each epoch within Δ time, ensuring synchronization. Additionally, Byzantine replicas cannot disrupt or halt the protocol during any epoch.

The epoch synchronization mechanism in BoundBFT relies on certificates: to start a new epoch, a certificate (i.e., block, blame, or equivocation certificate) must be formed in the previous epoch. BoundBFT ensures that, regardless of Byzantine behavior, a certificate is created in each epoch.

The mechanism BoundBFT employs to guarantee the existence of a certificate is as follows: honest replicas initiate a $timeoutCertificate$ upon entering a new epoch (line 2 in Algorithm 1). If the timeout expires without receiving a certificate, the replica blames the leader. This results in two possible outcomes: (i) an honest replica forms one of the certificates, or (ii) no honest replica receives a certificate before the $timeoutCertificate$ expires. In case (ii), all $f + 1$ honest replicas will blame the leader, resulting in the formation of a blame certificate.

Once a certificate is ensured for an epoch, synchronizing replicas becomes straightforward: each replica forwards the received certificate (lines 25 and 35 in Algorithm 1 and line 17 in Algorithm 2). Within Δ time, all honest replicas receive the certificate and start the next epoch if they have not already done so.

3.2.2 Intuition behind agreement

BoundBFT ensures that no two honest replicas commit different blocks in the same blockchain height. Consequently, the resulting blockchain remains consistent and does not have forks.

In epochs with a Byzantine leader, multiple certificates can be created. As a result, different honest replicas may start the next epoch receiving different certificates. For instance, one honest replica may receive a block certificate, while another may receive an equivocation certificate. To account for this scenario, an honest replica commits a proposed block b in epoch e only if it knows that the first certificate received by all honest replicas in e is a certificate for b. This guarantees two properties: (i) all honest replicas vote for block b, and (ii) all honest replicas lock on block b in epoch e. Property (i) ensures that no other block can be certified and afterward committed in epoch e. Property (ii) guarantees that honest replicas vote only for blocks extending b in the following epochs. As a result, only b and blocks extending b will be certified and committed in epochs $e' \geq e$, and the agreement property will be satisfied.

The mechanism BoundBFT uses to verify the commit condition is as follows. Upon receiving a certificate for block b, $C_e(b)$, as the first certificate in epoch e, r forwards the certificate and triggers $timeoutCommit(e)$ at time t (lines 32 and 35 in Algorithm 1). Consequently, r knows that all honest replicas will receive $C_e(b)$ by time $t + \Delta$. If an honest replica p received a different certificate before $C_e(b)$, it must have received it at time $t_1 < t + \Delta$. Since p also forwards its certificate, r will receive it by time $t_1 + \Delta < t + 2\Delta$. Therefore, setting $timeoutCommit(e)$ to 2Δ ensures that r receives p's certificate on time. Ultimately, if $timeoutCommit(e)$ expires and r has not heard about any other certificates, r can be sure that the first certificate received by all honest replicas in e is $C_e(b)$. In this case, r commits block b.

3.2.3 Intuition behind progress

BoundBFT ensures that all honest replicas commit a new block in every epoch with an honest leader. It does so by: (i) ensuring that all honest replicas vote for the leader's proposal, and (ii) preventing the creation of blame or equivocation certificates. Property (i) guarantees the creation of a unique block certificate, while property (ii) guarantees that all honest replicas must receive the block certificate, trigger $timeoutCommit$, and, when it expires, commit the proposed block.

An honest leader of an epoch proposes a new block that extends its $validBlock$ and sends $validBC$ together with the new block. Other honest replicas will vote for the new proposal only if the $validBC$ sent by the leader is at least as recent as their $lockedBC$. Consequently,

BoundBFT ensures that whenever an honest replica locks on a block in epoch e, all honest replicas update the *validBlock* and *validBC* to the block certified in epoch e. Therefore, the *validBC*s on all honest replicas are always at least as recent as *lockedBC*s on all honest replicas.

BoundBFT ensures that *validBlock* and *validBC* are always up to date by relying on a mechanism that uses $timeoutEpochChange$. Namely, an honest replica r cannot start the next epoch immediately if the first certificate it receives in the current epoch is a blame or equivocation certificate. Instead, it must ensure no other honest replica locks on a block in this epoch. Consequently, r forwards its certificate (line 17 in Algorithm 2), knowing that in Δ time, all honest replicas will receive it. If any honest replica p locked on a block, it must have done so before receiving the forwarded certificate. As a result, upon forwarding its certificate, r sets $timeoutEpochChange(e)$ to expire in 2Δ time (line 18 in Algorithm 2). Moreover, r starts the next epoch only when this timeout expires, or it receives the block certificate for the current epoch. Since p also forwards the certificate after locking (line 35 in Algorithm 1), r knows it will receive it before $timeoutEpochChange$ expires. Notably, r will not lock on a block certificate if it receives the certificate after $timeoutEpochChange$ is initiated.

Lastly, BoundBFT must ensure that no equivocation or blame certificates are possible in epochs with honest leaders. An equivocation certificate will not be formed since the honest leader will not propose two different blocks. However, ensuring that no blame certificate is possible requires that no honest replica blames the leader. In other words, every honest replica must receive the block certificate before $timeoutCertificate(e)$ expires. Consequently, honest replicas set $timeoutCertificate(e)$ to 3Δ. The first Δ accounts for epoch drift time, the second Δ for the time it takes for the leader's proposal to reach all honest replicas, and the last Δ is for the reception of the votes broadcast by honest replicas. Since we already showed that all honest replicas will vote for the honest leader, the block certificate is formed on all honest replicas before the $timeoutCertificate(e)$ expires, and no honest replica blames the leader.

4 Debunking synchrony violations

BoundBFT relies on synchrony every time it uses one of its timeouts, expressed as a multiple of a synchrony bound Δ. A synchrony violation may result in a scenario where the timeout expires before a replica receives an expected message from some honest replica. We refer to this phenomenon as a *timeout violation*. In this section, we first examine how malicious replicas may attempt to compromise BoundBFT. We then consider the consequences of timeout violations in the presence and absence of malicious replicas. We present a detailed Byzantine protocol in the appendix.

4.1 Byzantine behavior

Listing possible faulty behaviors of Byzantine replicas is unusual since, by definition, a Byzantine replica can behave arbitrarily. In the context of leader-based protocols where messages are signed, however, the scope for deviation is limited, as we now explain.

In the case of a Byzantine leader, these are the possible faulty behaviors:

- *SILENCE:* The leader does not send a proposal to a subset of replicas (possibly all).
- *EQUIVOCATION:* The leader proposes multiple blocks in the same epoch.

- *AMNESIA:* The leader does not extend the blockchain with a new block and proposes an alternative block for one of the previously committed blocks.

In addition, non-leader Byzantine replicas may misbehave as follows.

- *MULTI-VOTE:* A Byzantine replica can choose to vote for any proposal it likes. It can also vote for multiple proposals in the same epoch.
- *BLAME:* A Byzantine replica can blame the leader by broadcasting a blame message at any point in the execution, even if the leader is honest.

In addition, Byzantine replicas can always remain silent or discard messages selectively.

4.2 Timeout *timeoutCommit*

The *timeoutCommit* is the only timeout responsible for BoundBFT's agreement. An honest replica sets this timeout after locking on a block in an epoch (line 32 in Algorithm 1). The replica then forwards the block certificate and waits for this timeout to receive certificates from all other honest replicas.

If *timeoutCommit* is violated, an honest replica may miss a certificate from some honest replicas. As a result, it may commit block b in epoch e thinking all honest replicas locked on b, while in reality, some honest replicas received a different certificate and moved to epoch $e+1$ without locking on b. If enough honest replicas did not lock on b, the agreement might be compromised as honest replicas may vote for an alternative block b', create a block certificate, and commit b'.

The likelihood of this situation in the absence of Byzantine replicas is low, as the following conditions must be fulfilled:

1. A blame certificate must be formed in epoch e, meaning a majority of replicas must have blamed the leader in epoch e (i.e., $timeoutCertificate$ was violated in all of these replicas).
2. A majority of replicas did not lock on b in epoch e, receiving a blame certificate before receiving a block certificate for b.
3. The leader of epoch $e+1$ did not receive b's block certificate in epoch e, thus not updating its *validBlock* and *validBC* to b and b's certificate (i.e., its *timeoutEpochChange* was violated in epoch e).

Even though $timeoutCertificate$ and *timeoutEpochChange* are responsible for BoundBFT's progress, in this scenario, they also play a role in guarding the protocol's agreement.

Byzantine replicas can exploit *timeoutCommit* violations and potentially compromise BoundBFT's agreement through the following attacks:

- *AMNESIA-ATTACK*: The Byzantine leader ignores the algorithm (line 17 in Algorithm 1) and does not propose a block that extends its *validBlock*. Instead, it proposes an alternative block b for its *validBlock* (i.e., $b.prev = validBlock.prev$). Byzantine replicas vote for this proposal. The agreement can be violated if an honest replica committed *validBlock* while some honest replicas, due to *timeoutCommit* violations, did not lock on *validBlock*. As a result, these replicas will vote for block b, and if their votes, together with Byzantine votes, form a majority, block b will be certified and committed. To increase the probability of this scenario, in epochs with an honest leader, Byzantine replicas send votes for the block proposed by the honest leader to one subset of honest replicas to help them form the block certificate faster and commit a block. At the same time, they send BLAME messages to a different subset of honest replicas to help them form a blame certificate before receiving a block certificate.

- *EQUIVOCATION-ATTACK*: The Byzantine leader proposes two distinct proposals in the same epoch, and Byzantine replicas vote for both. The first proposal and its votes are sent to one subset of honest replicas, and the second proposal and its votes to another subset. As a result, two honest replicas may vote for, receive block certificates, and commit different blocks if their *timeoutCommit*s expire before they learn about the equivocated proposal. In epochs when the leader is honest, Byzantine replicas remain silent and update *validBlock* and *validBC* to stay aware of the most recently certified block. This is important so that when they become the leader, Byzantine replicas can generate new blocks that honest replicas will vote for.

4.3 Timeout $timeoutCertificate$

Honest replicas initiate this timeout upon starting an epoch (line 2 in Algorithm 2). Its purpose is twofold. First, it ensures that an honest replica does not wait indefinitely for a silent malicious leader. Second, when the leader is honest and proposes a block that all honest replicas will vote for, $timeoutCertificate$ should not expire before all honest replicas receive the proposal and the votes from all other honest replicas. In other words, before they receive a block certificate. Consequently, no honest replicas will blame an honest leader.

If $timeoutCertificate$ is violated, an honest replica will incorrectly blame the honest leader. If a majority of honest replicas blame the leader, a blame certificate can be formed, and the decision might not be reached in the current epoch. However, the block will be committed when the next honest leader proposes a block and other honest replicas receive the block certificate on time.

The creation of a blame certificate is easier in the presence of Byzantine replicas:

- *BLAME-ATTACK*: Byzantine replicas do not vote for the proposal sent by the honest leader. Instead, they broadcast BLAME messages upon starting the epoch with an honest leader. As a result, the blame certificate can be formed if a single honest replica blames the leader. In epochs when the leader is Byzantine, the Byzantine replicas just remain silent.

Apart from unconditionally blaming the leader and hoping that one of the honest replicas will also blame the leader, there is no other way for Byzantine replicas to prevent honest replicas from committing a block in epochs with honest leaders. $timeoutCertificate$ violations may slow down the execution but will not lead to violations in agreement (i.e., when two honest replicas decide on different blocks).

4.4 Timeout $timeoutEpochChange$

An honest replica r triggers this timeout when it receives a blame or equivocation certificate as the first certificate in an epoch (line 18 in Algorithm 2). This timeout ensures that if another honest replica receives a block certificate as the first certificate and locks on it in the same epoch, r will receive this block certificate and update its *validBlock* and *validBC* before starting the next epoch. Consequently, r starts the next epoch when it receives a block certificate from the current epoch or when $timeoutEpochChange$ expires.

If the $timeoutEpochChange$ is violated, an honest replica will not hear about the locked block. Consequently, if the replica is the next epoch leader, it will propose a block that locked replicas will not accept. If the set of remaining honest replicas that vote for a proposed block is less than the majority, the block certificate will not be formed, and a decision will not be reached even though the epoch leader is honest. However, the new block will be committed when one of the locked honest replicas becomes a leader.

Honest replicas rely on this timeout only in epochs when an equivocation or a blame certificate is formed. In the absence of attacks, creating an equivocation certificate is impossible. As a result, an honest replica uses *timeoutEpochChange* solely in case it receives a blame certificate. This can happen only if *timeout-Certificate* is violated on a majority of honest replicas, and they blame the honest leader.

With Byzantine replicas, however, both equivocation and blame certificates are possible. Byzantine replicas can exploit *timeoutEpochChange* violations as follows:

- *EQUIVOCATION-CERTIFICATE-ATTACK*: The Byzantine leader broadcasts a proposal for block b. Then, the Byzantine replicas send votes for block b to one subset of honest replicas to help them create a block certificate and lock on b. At the same time, the Byzantine leader sends a second proposal for block $b' \neq b$ to the other subset of honest replicas. These honest replicas will form an equivocation certificate and start *timeoutEpochChange*. As a result, they will not lock on block b.
- *BLAME-CERTIFICATE-ATTACK*: Similarly, Byzantine replicas impose locking on one subset of honest replicas by sending a proposal and votes for block b. Instead of equivocating, the Byzantine leader remains silent and does not send any proposal to the other subset of honest replicas. Moreover, all Byzantine replicas send blame messages to these replicas. If these replicas did not receive a block certificate before the *timeoutCertificate* expires,[1] they will blame the leader and, together with blame messages from Byzantine replicas, form a blame certificate and start *timeoutEpo-chChange*, without locking on block b.

In both attacks, Byzantine replicas remain silent in epochs with an honest leader. By remaining silent, these replicas ensure that if a *timeoutEpochChange* violation happened and the next honest leader proposes a block that does not extend block b, a block certificate will not be formed and a decision will not be reached.

Similarly to *timeoutCertificate*, violations of *timeoutEpochChange* can slow down the execution but cannot lead to violations in agreement.

5 Evaluation

5.1 Experimental environment and setup

We conducted our experiments in a cluster with emulated wide-area latencies between 6 AWS zones (see Table 1). Latencies between nodes were configured using the Linux Traffic Control kernel module [25]. The emulated WAN provided an affordable approximation of the AWS environment since our evaluation required hundreds of hours of experiments (see Appendix B). The cluster contains 60 nodes divided in two groups: (i) EPYC Zen 2 with two 16-Core AMD EPYC 2881 MHz and 32GB of RAM, and (ii) HP SE1102 with two Quad-Core Intel Xeon 2.5GHz and 8GB of RAM. We implemented BoundBFT, all competing protocols, and all proposed attacks (see the Appendix) in Go. The implementations use SHA256 hashes and Ed25519 64-byte digital signatures. We rely on libp2p [1] for communication between replicas.

[1] Even though the Byzantine replicas do not send the proposal and votes to these replicas, they can receive the forwarded messages from other honest replicas and form a block certificate before *timeoutCertificate* expires.

5.2 BoundBFT's synchrony bound

In this section, we experimentally determine the value for BoundBFT's synchrony bound Δ: We ran BoundBFT in the presence of malicious replicas and determined Δ that gives enough confidence that BoundBFT's correctness will not be compromised. Initially, we set Δ to 1250 ms (99.99%), the synchronous bound from [30], and gradually decreased it to the point where we started to observe BoundBFT's agreement and progress violations. The complete data for these experiments can be found in Appendix B. In the following, we comment on the main takeaways.

When there is a single ($f = 1$) or no ($f = 0$) Byzantine replicas in the system, we did not observe any agreement violations, even if we set Δ as low as 50 ms—the average latency between 80% of replicas in our system is higher than 50 ms. This shows that agreement violations are highly unlikely if the number of Byzantine replicas is low, even if many messages violate synchronous bounds. When the number of Byzantine replicas is $f = 19$ (i.e., the maximum number of Byzantine replicas partially synchronous protocols can tolerate), the agreement violations were observed only when we lowered the Δ to 50 ms. Moreover, even with $\Delta = 50$ ms, the BoundBFT's agreement was violated in less than 10% of epochs in which the attack was launched. However, to prevent agreement violations when the number of Byzantine replicas is $f = 29$ (i.e., the maximum BoundBFT can tolerate), we needed to increase Δ to 150ms and 300ms, for 1KB and 32KB block sizes, respectively. This makes sense since to create a block certificate an honest replica needs to receive a vote from itself, Byzantine replicas, and one honest replica. Importantly, even in this case, the resulting Δ was 8 and 4 times lower than the initial one. Notice that when $f = 29$, partially synchronous protocols will halt if Byzantine replicas remain silent.

Table 2 shows the Δs that resulted in no agreement violations and less than 5% of progress violations for all considered setups. Namely, no two honest replicas committed on different blocks for the same height and in less than 5% of epochs with an honest leader, some honest replicas did not commit a new block. The progress violations can also be lowered to 0 %, but this would require a slight increase in the Δ chosen (see Appendix B). We believe this is not necessary since progress violations can only lead to a slight decrease in performance and do not affect agreement.

Attack type	1KB						32KB					
	f=29		f=19		f=1		f=29		f=19		f=1	
	k_{min}	k_{max}	k_{min}	k_{max}	k_{min}	k_{max}	k_{min}	k_{max}	k_{min}	k_{max}	k_{min}	k_{max}
EQUIVOCATION	150	150	100	100	100	100	150	300	150	150	100	100
AMNESIA	150	150	150	150	100	100	300	300	150	150	100	100
EQUIVOCATION-CERTIFICATE	150	150	150	150	100	100	300	300	150	150	100	100
BLAME-CERTIFICATE	150	150	150	150	100	100	150	150	150	150	100	100
BLAME	150		150		100		300		150		100	
NO ATTACK	100						100					

Table 2 The Δ in ms BoundBFT must adopt to achieve 0% of Agreement and $< 5\%$ of Progress violations under a specific attack. The table shows data for the setup of 60 replicas, 1KB and 32KB block sizes and different number of Byzantine replicas (f). Additionally, for attacks that partition honest replicas in two subsets, it shows results when Byzantine replicas divide honest replicas into the two smallest subsets ($k = k_{min} = 1$) and the two largest subsets ($k = k_{max} = n - f/2$).

5.3 Performance

In this section, we compare BoundBFT to state-of-the-art partially synchronous and synchronous protocols. In the partially synchronous model, we choose Tendermint [5] and HotStuff-2 [32], the most recent protocol of the HotStuff family [39]. In the synchronous system model, we consider Sync HotStuff [2, 3]. We limited our evaluation to Byzantine

consensus protocols with a rotating leader: a new leader is elected when the protocol changes epoch (or round or view) as part of the normal execution, not only when the leader fails. These protocols are preferred in blockchain systems since they provide better fairness and censorship resistance than protocols with a stable leader (e.g., PBFT) [3].

To generate a load in our experiments, we equip every replica with a built-in client that generates transactions in advance and stores them in a local pool. When a replica is a leader in an epoch, it takes transactions from the pool and forms a block. The block size defines the number of transactions taken from the pool. This design leaves the mempool (i.e., the part of a blockchain responsible for propagating client transactions across the system) out of the discussion as different systems may implement it in different ways. Consequently, the latencies we report in the paper represent consensus latencies (i.e., the time the leader of an epoch needs to commit a block). Throughput is computed as the rate of committed blocks per time unit. Every point in the graphs is an average of 3 runs. We ran each experiment for 5 minutes.

5.3.1 Latency

BoundBFT and Sync HotStuff wait for 2Δ (BoundBFT's $timeoutCommit(e)$) before committing a block in epoch e. Consequently, their latency is directly affected by the chosen synchrony bound. For BoundBFT, we adopt the synchrony bound based on the experiments from the previous section (see Table 2). Conversely, for Sync HotStuff, we use the Δ from [30].

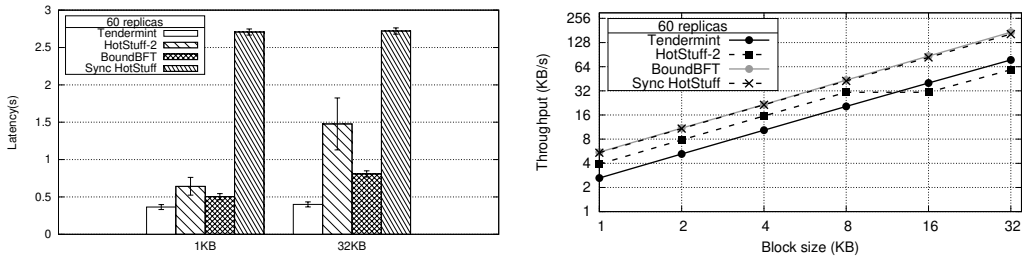

Figure 2 Latency (left) and throughput (right) comparison for all protocols for 1 KB and 32 KB block sizes in a system with 60 replicas.

We measured latencies in a system with 60 replicas with 1KB and 32KB block sizes. Figure 2 (left) shows the average latency computed by epoch leaders. First, we notice the significant impact of BoundBFT's synchrony bound on latency. Namely, BoundBFT achieves 5.4× and 3.4× lower latency than Sync HotStuff with 1KB and 32KB block sizes, respectively. Second, BoundBFT's latency is in between the latencies of partially synchronous protocols. It is 1.3× and 1.8× lower than HotStuff-2's latency and 1.4× and 2× higher than Tendermint's for small and large blocks, respectively. HotStuff-2 has higher latency due to its linear communication pattern, which requires five communication steps, while Tendermint has quadratic communication and commits a block in only three communication steps. We can also see that HotStuff-2 has the most significant standard deviation; we attribute this to the lower redundancy due to its linear communication pattern.

5.3.2 Throughput

BoundBFT and Sync HotStuff [2, 3] use *pipelining* to limit the impact of Δ on throughput, which allows the leader to propose a block B_{k+1} that extends block B_k after receiving

$C_e(B_k)$, i.e., before committing block B_k. This way, protocols can order multiple blocks in parallel, and the throughput is unaffected by Δ. This technique was initially introduced in HotStuff [39], and we implemented a pipelined version of HotStuff-2. Adapting pipelining to Tendermint is more complex and out of the scope of this paper. Moreover, notice that we compare BoundBFT a pipelined partially synchronous protocol, HotStuff-2.

We evaluated throughout in a system with 60 replicas and various block sizes (see Figure 2 (right)). BoundBFT and Sync HotStuff have similar throughout, as they both start ordering the next block after receiving a certificate for the previous block. Moreover, they outperform partially synchronous protocols for all block sizes cosidered, reaching throughput more than 2× higher than Tendermint's for all block sizes. The reason behind this is that Tendermint does not use pipelining. They also perform better, from 1.4× to 3×, than HotStuff-2, a partially synchronous protocol with pipelining. This is because even though both protocols start ordering the next block after collecting a certificate for the previous block, the certificate in HotStuff-2 requires votes from a two-third majority of replicas, while in BoundBFT the votes from the majority are enough.

5.4 Summary

In this section, we summarize the main takeaways of our evaluation.

- We did not observe any agreement violations when the number of Byzantine replicas was 0 and 1. Synchrony violations combined with a minority or one-third of colluded Byzantine replicas resulted in agreement violations.
- BoundBFT can use a Δ that is 4× to 8× smaller than the conservative 99.99% Δ [30], allowing it to improve latency from $\approx$ 3.4× to 5.4×.
- BoundBFT's Δ is big enough to ensure correctness with high probability when the system is under attack.
- BoundBFT achieves from 1.4× to 3× higher throughput and comparable latency to state-of-the-art partially synchronous protocols.

6 Related work

The synchronous system model in its purest form requires that every message sent in the system be delivered within some known synchrony bound Δ. Some protocols have been designed for this model such as Dfinity [24] and Sync HotStuff [2, 3]. These protocols are optimal in terms of resilience [15, 26] as they can tolerate up to a minority of faulty processes (i.e., $f < n/2$, where f is the number of Byzantine processes out of n). However, deploying these protocols in a blockchain environment usually comes with conservative Δ that should ensure with high probability that the system is synchronous. Consequently, these protocols perform poorly. BoundBFT belongs to this family of protocols and is similar to the rotating version of Sync HotStuff [3]. The two protocols share similar behavior in the common-case but they have different epoch synchronization mechanisms. In this paper, we show that we do not need to use conservative bounds when deploying BoundBFT and as a consequence it can deliver reasonably good performance.

Many deterministic consensus protocols assume the partially synchronous system model [13]. The model allows consensus protocols that need synchrony only for liveness but not for safety. More precisely, the partially synchronous system model ensures that after some point, usually referred to as GST (Global Stabilization Time), messages exchanged among honest processes will be delivered within an unknown synchrony bound. A fundamental limitation

of the partially synchronous system model is that any consensus protocol designed for this model can tolerate up to one third of faulty processes (i.e., $f < n/3$) [13]. The first practical representative of these protocols is PBFT [7], and more recently, in the blockchain context, Tendermint [5], HotStuff [39], HotStuff-2 [32] and ICC [6], to name a few.

Guo et al. [23] introduced the "weak synchronous model" (called mobile sluggish model in [2] for consistency with other works in the literature). The idea is to distinguish between *prompt* processes, those that respect synchrony bounds, from *sluggish* processes, those for whom messages may violate synchrony bounds. Moreover, the set of sluggish processes may change over time. The mobile sluggish model weakens the synchronous model and may result in more practical protocols. However, this is true only in situations when the actual number of faulty processes in the system is less than the minority (e.g., [2, 8]).

XFT [30] is based on the observation that typical BFT consensus protocols assume a powerful adversary that fully controls malicious processes and the network between honest processes, which is unrealistic. We share this view, and consider an adversary that cannot control the network between honest processes. XFT differentiates between three types of faulty processes: crash, Byzantine, and partitioned (i.e., processes that cannot exchange messages with other honest processes within the known synchrony bound). It ensures progress as long as the total number of faulty processes in the system is lower than $f < n/2$. In other words, XFT assumes a majority of honest replicas that can communicate timely. Since selecting a quorum of $f + 1$ responsive replicas out of n replicas requires an exponential number of attempts, the solution is practical when f is small.

An alternative way to increase the resilience of consensus protocols while assuming a partially synchronous system is to rely on trusted hardware (e.g., Trusted Execution Environment). This idea was introduced in A2M [9] and explored in many works (e.g, [10, 29, 37, 29, 38, 11]). These techniques have the disadvantage of requiring blockchain servers to be equipped with special hardware, which is not the case with BoundBFT.

The consensus protocols presented above assume some synchrony to circumvent the FLP impossibility result [16]. Alternatively, there are many asynchronous consensus protocols that rely on randomization to solve consensus [4, 34, 14, 20, 36, 33, 22, 18, 21]. Asynchronous protocols are robust since they do not assume any synchrony, but they provide probabilistic guarantees and perform worse than partially synchronous and synchronous protocols. Consequently, some protocols use a simpler leader-based deterministic protocol to improve the latency in good cases [19, 27, 35, 31].

7 Conclusion

In this paper, we have shown how we can circumvent major performance drawbacks of synchronous consensus protocols by choosing protocol-specific time bounds instead of conservative model-specific bounds. Instead of ensuring that all messages are received within synchronous bounds with high probability, we have analyzed protocol semantics and potential correctness violations in case of synchrony violations and Byzantine attacks, and have shown how to select a time bound that does not hurt correctness. As a showcase, we designed BoundBFT, a new Byzantine fault-tolerant synchronous consensus protocol, and have shown experimentally that BoundBFT withstands synchrony bound violations under attack and outperforms traditional synchronous consensus protocols. Furthermore, BoundBFT achieves similar latency and better throughput than state-of-the-art partially synchronous protocols, offering higher resilience.

References

1. Libp2p. https://libp2p.io. [Accessed 2023-01-12].
2. Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, may 2020. doi:10.1109/sp40000.2020.00044.
3. Ittai Abraham, Kartik Nayak, and Nibesh Shrestha. Optimal good-case latency for rotating leader synchronous BFT. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, pages 27:1–27:19, 2022. doi:10.4230/LIPICS.OPODIS.2021.27.
4. Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, PODC'83, pages 27–30. ACM, aug 1983. doi:10.1145/800221.806707.
5. Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. jul 2018. URL: http://arxiv.org/abs/1807.04938, arXiv:1807.04938[cs.DC], doi:10.48550/arXiv.1807.04938.
6. Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet computer consensus. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, PODC'22, pages 81–91, jul 2022. doi:10.1145/3519270.3538430.
7. Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, OSDI '99, pages 173–186, USA, 1999. USENIX Association. URL: https://dl.acm.org/citation.cfm?id=296824.
8. T-H. Hubert Chan, Rafael Pass, and Elaine Shi. Pili: An extremely simple synchronous blockchain. Cryptology ePrint Archive, Paper 2018/980, 2018. https://eprint.iacr.org/2018/980. URL: https://eprint.iacr.org/2018/980.
9. Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, volume 41, pages 189–204. ACM, oct 2007. doi:10.1145/1294261.1294280.
10. Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil*, pages 174–183. IEEE Computer Society, 08 2004. doi:10.1109/RELDIS.2004.1353018.
11. Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. DAMYSUS: streamlined bft consensus leveraging trusted components. In *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, mar 2022. doi:10.1145/3492321.3519568.
12. Danny Dolev and H. Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12:656–666, 11 1983. doi:10.1137/0212045.
13. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, apr 1988. doi:10.1145/42282.42283.
14. Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, aug 1997. doi:10.1137/s0097539790187084.
15. Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Comput.*, 1(1):26–39, jan 1986. doi:10.1007/BF01843568.

16 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, apr 1985. doi:10.1145/3149.214121.

17 Matthias Fitzi. *Generalized communication and security models in Byzantine agreement*. PhD thesis, ETH Zurich, Zürich, Switzerland, 3 2003. Reprint as vol. 4 of ETH Series in Information Security and Cryptography, ISBN 3-89649-853-3, Hartung-Gorre Verlag, Konstanz, 2003. URL: https://d-nb.info/967397375.

18 Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-ng: Fast asynchronous BFT consensus with throughput-oblivious latency. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, CCS '22, pages 1187–1201, New York, NY, USA, 2022. ACM. doi:10.1145/3548606.3559379.

19 Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. *CoRR*, abs/2106.10362, 06 2021. URL: https://arxiv.org/abs/2106.10362, arXiv:2106.10362, doi:10.48550/arXiv.2106.10362.

20 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, Shanghai, China, October 28-31, 2017*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM. doi:10.1145/3132747.3132757.

21 Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Speeding dumbo: Pushing asynchronous bft closer to practice. Cryptology ePrint Archive, Paper 2022/027, 2022. https://eprint.iacr.org/2022/027. URL: https://eprint.iacr.org/2022/027.

22 Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous BFT protocols. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, CCS '20, pages 803–818, New York, NY, USA, 2020. ACM. doi:10.1145/3372297.3417262.

23 Yue Guo, Rafael Pass, and Elaine Shi. Synchronous, with a chance of partition tolerance. In *Advances in Cryptology – CRYPTO 2019*, pages 499–529, aug 2019. doi:10.1007/978-3-030-26948-7_18.

24 Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, may 2018. URL: http://arxiv.org/abs/1805.04548, arXiv:1805.04548, doi:10.48550/arXiv.1805.04548.

25 Bert Hubert, Gregory Maxwell, Martijn van Oosterhout, Remco van Mook, Paul B. Schroeder, et al. Linux advanced routing & traffic control HOWTO. https://lartc.org/lartc.html, 2002. [Accessed 2024-22-5].

26 Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference*, volume 4117 of *Lecture Notes in Computer Science*, pages 445–462. Springer, 2006. URL: https://iacr.org/archive/crypto2006/41170440/41170440.pdf, doi:10.1007/11818175_27.

27 Klaus Kursawe and Victor Shoup. Optimistic asynchronous atomic broadcast. *IACR Cryptol. ePrint Arch.*, page 22, 2001. https://eprint.iacr.org/2001/022. URL: http://eprint.iacr.org/2001/022.

28 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, jul 1982. doi:10.1145/357172.357176.

29 Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *NSDI*, volume 9, pages 1–14, 2009. URL: http://www.usenix.org/events/nsdi09/tech/full_papers/levin/levin.pdf.

684 **30** Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT:
685 practical fault tolerance beyond crashes. In Kimberly Keeton and Timothy Roscoe, editors,
686 *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Sa-*
687 *vannah, GA, USA, November 2-4, 2016*, OSDI'16, pages 485–500, USA, 2016. USENIX
688 Association. URL: `https://www.usenix.org/conference/osdi16/technical-sessions/`
689 `presentation/liu`.
690 **31** Yuan Lu, Zhenliang Lu, and Qiang Tang. Bolt-dumbo transformer: Asynchronous consensus
691 as fast as pipelined BFT. *CoRR*, abs/2103.09425, 2021. URL: `https://arxiv.org/abs/2103.`
692 `09425`, `arXiv:2103.09425`, `doi:10.48550/arXiv.2103.09425`.
693 **32** Dahlia Malkhi and Kartik Nayak. Extended abstract: Hotstuff-2: Optimal two-phase responsive
694 BFT. *IACR Cryptol. ePrint Arch.*, page 397, 2023. URL: `https://eprint.iacr.org/2023/`
695 `397`.
696 **33** Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT
697 protocols. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers,
698 and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and*
699 *Communications Security, Vienna, Austria, October 24-28, 2016*, CCS '16, pages 31–42, New
700 York, NY, USA, 2016. ACM. `doi:10.1145/2976749.2978399`.
701 **34** Michael O. Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations*
702 *of Computer Science (sfcs 1983)*, nov 1983. `doi:10.1109/sfcs.1983.48`.
703 **35** HariGovind V. Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-
704 tolerant atomic broadcast. *IACR Cryptol. ePrint Arch.*, page 82, 2006. `https://eprint.iacr.`
705 `org/2006/082`. URL: `http://eprint.iacr.org/2006/082`.
706 **36** Dmitry Tanana. Avalanche blockchain protocol for distributed computing security. In *2019*
707 *IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*.
708 IEEE, jun 2019. `doi:10.1109/blackseacom.2019.8812863`.
709 **37** Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo
710 Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30,
711 2013. `doi:10.1109/TC.2011.221`.
712 **38** Sravya Yandamuri, Ittai Abraham, Kartik Nayak, and Michael K. Reiter. Communication-
713 efficient bft protocols using small trusted hardware to tolerate minority corruption. Cryptology
714 ePrint Archive, Paper 2021/184, 2021. `https://eprint.iacr.org/2021/184`. URL: `https:`
715 `//eprint.iacr.org/2021/184`.
716 **39** Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham.
717 HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019*
718 *ACM Symposium on Principles of Distributed Computing*, PODC'19, pages 347–356, jul 2019.
719 `doi:10.1145/3293611.3331591`.

A Appendix: Algorithm correctness

A.1 Proof of correctness

This section presents the proof that BoundBFT satisfies all the properties of blockchain consensus protocol (Section 2.2).

▶ **Lemma 1.** *Every honest replica always moves to the next epoch.*

Proof. Assume for contradiction that an honest replica r remains in some epoch e indefinitely. This would imply that r did not generate any of the certificates $C_e(B_k)$, $C_e(\text{BLAME})$, or $C_e(\text{EQUIV})$. However, each honest replica starts the timer, $timeoutCertificate(e)$, upon entering epoch e (line 2 in Algorithm 2). When this timeout expires, if an honest replica has not received any certificate, it broadcasts the BLAME message (lines 3–5 in Algorithm 2). Consequently, if no certificate is formed before the $timeoutCertificate(e)$ expires, all honest replicas will broadcast the BLAME message, leading to the formation of the blame certificate $C_e(\text{BLAME})$. This contradicts our assumption, thus proving that every honest replica moves to the next epoch. ◀

▶ **Lemma 2.** *If an honest replica starts epoch e at time t, then all honest replicas start epoch e by time $t + \Delta$.*

Proof. Suppose an honest replica r starts epoch e at time t. This implies that r receives and broadcasts $C_{e-1}(B_k)$ at time t (lines 35–36 in Algorithm 1), or at time $t -$ timeoutEpochChange(2Δ), r receives and broadcasts $C_{e-1}(\text{BLAME})$ or $C_{e-1}(\text{EQUIV})$ (lines 17 and 21 in Algorithm 2). Messages with certificates will arrive within Δ time. Consequently, in the former case, all honest replicas receive $C_{e-1}(B_k)$ by time $t + \Delta$ and start epoch e. In the latter case, all honest replicas receive $C_{e-1}(\text{BLAME})$ or $C_{e-1}(\text{EQUIV})$ by time $t - \Delta$ and within 2Δ they start epoch e, ensuring that all honest replicas start epoch e by time $t + \Delta$. ◀

▶ **Theorem 3.** *(Epoch synchronization) All honest replicas continuously move through epochs, with each replica starting a new epoch within Δ time of any other honest replica.*

Proof. We prove this theorem by combining Lemma 1 and Lemma 2.

First, from Lemma 1, we know that every honest replica always moves to the next epoch. This ensures that no honest replica remains stuck in any epoch indefinitely.

Second, from Lemma 2, we know that if an honest replica starts epoch e at time t, then all honest replicas start epoch e by time $t + \Delta$. This guarantees that all honest replicas start each epoch within Δ time of each other.

Combining these two results, we can conclude that all honest replicas continuously move through epochs, with each replica initiating a new epoch within Δ time of any other honest replica. ◀

▶ **Lemma 4.** *If an honest replica directly commits block B_k in epoch e, then (i) no block different than B_k can be certified in epoch e, and (ii) every honest replica locks on block B_k in epoch e.*

Proof. Suppose an honest replica r directly commits B_k in epoch e at time t (line 40 in Algorithm 1). This means that at time $t - 2\Delta$, r received $C_e(B_k)$, locked on it, and started $timeoutCommit(e)$ (lines 26–32 in Algorithm 1). Moreover, replica r forwarded all messages representing $C_e(B_k)$ (lines 25 and 35 in Algorithm 1) so all honest replicas received these messages in Δ time, by time $t - \Delta$.

For part (i), assume for a contradiction that some honest replica p received and voted in epoch e for the block $B_l \neq B_k$. Since every honest replica votes only once, p must have received a proposal for B_l before receiving a proposal message for B_k, at some time $t_1 < t - \Delta$. As a result, p forwards the propose message for B_l at time t_1 (line 25 in Algorithm 1). Replica r will receive this message by time $t_1 + \Delta$, that is, before t. Since these two propose messages lead to a $C_e(\text{EQUIV})$ certificate, p would not commit (lines 9–12 and 15 in Algorithm 2), a contradiction. Therefore, property (i) holds since no honest replica votes for a block different than B_k; otherwise replica r would not commit.

For part (ii), we know by (i) that if replica r directly commits in epoch e, there is not any possible $C_e(B_l) \neq C_e(B_k)$. So, we need to prove that every honest replica p receives $C_e(B_k)$ before receiving $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$. For a contradiction, assume that p receives $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ before receiving $C_e(B_k)$. This must happen at time $t_1 < t - \Delta$ as p receives $C_e(B_k)$ by time $t - \Delta$. After receiving $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$, p broadcasts them (line 17 in Algorithm 2). So, p broadcasts $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ at time t_1 and r receives them by time $t_1 + \Delta$. Since $t > t_1 + \Delta$ replica r will not commit B_k, a contradiction. ◀

▶ **Lemma 5.** *If B_k is the only certified block in epoch e and $f + 1$ honest replicas lock on block B_k in epoch e ($lockedBlock = B_k$ and $lockedBC = C_e(B_k)$), then in all epochs $e' > e$, they vote only for blocks extending B_k, or they blame the proposer.*

Proof. The proof proceeds by induction on the epoch number.

Base step ($e' = e + 1$) : Let C denote the set of $f + 1$ honest replicas. The replicas in set C do not vote for proposals that do not extend blocks certified in epochs higher than or equal to their *lockedBC* (line 22 in Algorithm 1). As a result, when $timeoutCertificate(e')$ expires, no block certificate will be formed since no honest replica has voted, causing honest replicas to blame the proposer by sending $\langle \text{BLAME}, e' \rangle_*$ message. Therefore, the lemma holds for the base step since honest replicas vote only for a block if it extends *lockedBlock*.

Induction step ($e' \rightarrow e' + 1$): Assume that no replica in set C has voted for a block not extending B_k until epoch $e' + 1$. We now show that the lemma holds for epoch $e' + 1$. Since replicas from the set C vote for blocks extending B_k or blame the proposer in epochs $e \leq e'' \leq e'$, no block B_l not extending B_k can receive $f + 1$ votes in those epochs. Therefore, for all processes in set C, $lockedBlock = B_{k'}$ and $lockedBC.epoch \geq e$, where $B_{k'} = B_k$ or $B_{k'}$ extends B_k. Assume, for the sake of contradiction, that a process p in set C votes in epoch $e' + 1$ for a block not extending B_k. An honest replica will not vote for a block not extending its *lockedBlock* (line 22 in Algorithm 1), leading to a contradiction. Hence, the lemma holds for epoch $e' + 1$ as well. ◀

▶ **Lemma 6.** *If an honest replica directly commits block B_k in epoch e, then any block B_l that is certified in epoch $e' > e$ must extend B_k.*

Proof. The proof follows directly from Lemmas 4 and 5. More precisely, if an honest replica directly commits block B_k in epoch e, by Lemma 4, we know that $f + 1$ honest replicas (set C) lock on block B_k in epoch e and B_k is the only certified block in epoch e. Consequently, by Lemma 5, replicas from C vote only for the blocks extending block B_k in epochs $e' > e$. Therefore, no block B_l that does not extend B_k can collect $f + 1$ votes and thus cannot be certified in any epoch $e' > e$. ◀

▶ **Theorem 7.** *(Agreement) No two honest replicas commit different blocks at the same height.*

Proof. Suppose, for the sake of contradiction, that two distinct blocks B_k and B'_k are committed for the same height k. Assume that B_k is committed as a result of B_l being directly committed in epoch e and B'_k is committed as a result of $B_{l'}$ being directly committed in epoch e'. Without loss of generality, assume $l < l'$. Note that all directly committed blocks are certified. This is true because in order to start $timeoutCommit(e)$ for block B_k, a replica needs to receive $C_e(B_k)$ (lines 26–32 in Algorithm 1). By Lemma 6, $B_{l'}$ extends B_l. Therefore, $B_k = B'_k$, which contradicts the assumption that B_k and B'_k are distinct. Hence, no two honest replicas can commit different blocks at the same height. ◀

▶ **Lemma 8.** *If an honest replica r locks on a block B_k in epoch e, no honest replica starts epoch $e+1$ before updating $validBC$ to $C_e(B_l)$, where B_l does not need to be equal to B_k.*

Proof. Assume that an honest replica r locks on a block B_k in epoch e at time t. This implies r receives $C_e(B_k)$ at time t and does not receive $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ before that. Since all messages representing $C_e(B_k)$ are broadcast (lines 25 and 35 in Algorithm 1), all honest replicas receive these messages by time $t + \Delta$.

Suppose for a contradiction that some honest replica p starts the epoch $e+1$ before receiving $C_e(B_k)$ or some other $C_e(B_l)$, in other words at time $t_1 < t + \Delta$. This means it had received $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ and broadcast messages representing them at time $t_1 - 2\Delta$ (line 17 in Algorithm 2). Consequently, replica r receives $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ by time $t_1 - \Delta$ and as $t > t_1 - \Delta$, it does not lock on B_k, a contradiction. ◀

▶ **Corollary 9.** *Every honest replica starts epoch e with $validBC$ that is at least as recent as any certificate any honest replica locks on in any epoch $e' < e$.*

Proof. Suppose that the last epoch in which some honest replica locks on a block is epoch $e' < e$. By Lemma 8, we know that all honest replicas update their $validBC$ to some certificate from the same epoch (e'), before starting epoch $e' + 1$. From this and the fact that no honest replica, in any of the following epochs ($e' < e'' < e$), updates its $validBC$ to an older certificate (lines 33–34 in Algorithm 1), we see that this corollary holds. ◀

▶ **Theorem 10.** *(Progress) All honest replicas keep committing new blocks.*

Proof. From Lemmas 1 and 2, we see that replicas proceed through epochs, each epoch having a dedicated leader. If the leader of an epoch is Byzantine and does not propose any block or proposes equivocating blocks, honest replicas will collect $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ and move to the next epoch. Due to the round-robin leader election, there will be epochs with honest leaders.

Consider an epoch e with an honest leader l. Let t be the time when the first honest replica starts epoch e. By Lemma 2, all honest replicas enter epoch e by the time $t + \Delta$. Therefore, by the time $t + \Delta$ at the latest, an honest leader l broadcasts the proposal $\langle \text{PROPOSE}, e, B_k, validBC_l \rangle_l$. All honest replicas receive proposal by time $t + 2\Delta$. Since by the Corollary 9, $validBC_l$ is at least as recent as any $lockedBC$ of any honest replica, all honest replicas vote for the proposal. As a result, all honest replicas receive $C_e(B_k)$ by time $t + 3\Delta$. Since $timeoutCertificate(e) > 3\Delta$, no honest replica will send a $\langle \text{BLAME}, e \rangle_*$ message in epoch e, and $C_e(\text{BLAME})$ cannot be formed. Furthermore, considering that replica l is honest, it does not equivocate, so no $C_e(\text{EQUIV})$ will be formed in epoch e. Consequently, all honest replicas start $timeoutCommit(e)$, and when it expires, they commit B_k and all its ancestors. This scenario will occur in every epoch with an honest leader, ensuring that all honest replicas consistently commit new blocks across all such epochs. ◀

▶ **Theorem 11.** *(External validity) Every committed block satisfies the predefined* valid() *predicate.*

Proof. This follows directly from the requirement that every committed block must first be certified (lines 26, 32, and 40 in Algorithm 1). This implies that at least one honest replica accepted the block, meaning that *valid*() returned true for this block on at least one honest replica (line 19 in Algorithm 1). ◀

A.2 The Byzantine protocol

Algorithm 3 The Byzantine protocol

```
1: Initialization:
2:   e_p := 0                                                          ▷ the current epoch
3:   validBC_p := nil                          ▷ the most recent block certificate the replica is aware of and...
4:   validBlock_p := nil                                  ▷ the block certified by validBC_p
5:   C := getAllHonestReplicas()                          ▷ the set of honest replicas
6:   f := getNumberOfByzantineReplicas()                  ▷ the number of Byzantine replicas
7:   attackType := getAttackType()                        ▷ the attack type the Byzantine replica launches
8:   k := getTargetSize()        ▷ the size of the two random sets of honest replicas that are under the attack
9:  when bootstrapping do StartEpoch(0)                   ▷ the execution starts in epoch 0
10: Procedure StartEpoch(e) :                             ▷ upon starting the epoch...
11:   e_p ← e                                             ▷ the replica sets the current epoch, and...
12:   switch attackType :          ▷ invokes the specific attack and pass the necesarry arguments to it
13:     case EQUIVOCATION-ATTACK :
14:       LaunchEquivocationAttack(e, validBlock, validBC, C, k, f)
15:     case AMNESIA-ATTACK :
16:       LaunchAmnesiaAttack(e, validBlock, C, k, f)
17:     case BLAME-ATTACK :
18:       LaunchBlameAttack(e, C, f)
19:     case EQUIVOCATION-CERTIFICATE-ATTACK :
20:       LaunchEquivocationCertificateAttack(e, validBlock, validBC, C, k, f)
21:     case BLAME-CERTIFICATE-ATTACK :
22:       LaunchBlameCertificateAttack(e, validBlock, validBC, C, k, f)
23: when receive ⟨PROPOSE, e, b, BC⟩_l and f + 1 distinct ⟨VOTE, e, id(b)⟩_*   ▷ when replica receives a proposal...
24:  where e = e_p do                                     ▷ and f + 1 votes for it in the current epoch...
25:   cert ← NewCert from f + 1 ⟨VOTE, e, id(b)⟩_*         ▷ it forms a block certificate,...
26:   validBC_p ← cert                                    ▷ updates its validBC_p and validBlock_p...
27:   validBlock_p ← b                                    ▷ to the most recent block, and...
28:   StartEpoch(e + 1)                                   ▷ starts immidiately the next epoch
29: when receive ⟨PROPOSE, e, b, BC⟩_p and ⟨PROPOSE, e, b′, BC′⟩_p   ▷ upon receiving two different proposals...
30:  where e = e_p and p = leader(e) and b ≠ b′ do         ▷ from the leader in the current epoch:
31:   start timeoutEpochChange(e_p)                       ▷ the replica triggers timeoutEpochChange
32: when receive f + 1 distinct ⟨BLAME, e⟩_* where e = e_p do   ▷ upon receiving a blame certificate in the...
33:   start timeoutEpochChange(e_p)          ▷ current epoch the replica triggers timeoutEpochChange
34: when timeoutEpochChange(e) expires do                 ▷ when timeoutEpochChange expires and...
35:   if e = e_p then                                     ▷ the replica is stil in epoch e...
36:     StartEpoch(e_p + 1)                               ▷ the replica starts the next epoch
```

Algorithm 3 presents the Byzantine replica protocol. Byzantine replicas proceed through epochs in the same way as honest replicas. Namely, if they receive a block certificate, they start the next epoch immediately (lines 23–28 in Algorithm 3), while if they receive a blame or equivocation certificate they wait for *timeoutEpochChange* before starting the next epoch (lines 29–36 in Algorithm 3). They do this to be synchronized with honest replicas so they can launch the attack at the moment that maximizes the attack's effectiveness. Moreover, a Byzantine replica waits for *timeoutEpochChange* to update its *validBlock* and *validBC* to the most recent values. As a result, when leader in an epoch, a Byzantine replica can propose a valid block (i.e., an invalid block would be easily dismissed by honest replicas).

Algorithms 4 and 5 present the logic for attacks on BoundBFT's agreement and progress, respectively. We empower the attacks by assuming that Byzantine replicas know each other and collude (Section 2.1): each Byzantine replica has private keys of all Byzantine replicas.

Therefore, a Byzantine replica can sign and send messages on behalf of other Byzantine replicas.

Upon starting an epoch, a Byzantine replica launches a specific attack, which is a parameter of an algorithm:

- *EQUIVOCATION-ATTACK*,
- *AMNESIA-ATTACK*,
- *BLAME-ATTACK*,
- *EQUIVOCATION-CERTIFICATE-ATTACK*, or
- *BLAME-CERTIFICATE-ATTACK*.

Algorithm 4 Byzantine attacks on BoundBFT's agreement

```
1:  Procedure LaunchEquivocationAttack(                          ▷ ***EQUIVOCATION-ATTACK***
2:   e, validBlock, validBC, honestReplicas, k, f)
3:     if isByzantineLeader(e) then                              ▷ if the epoch leader is a Byzantine replica:
4:       p1, p2 ← generateTwoDifferentProposals(e, validBlock, validBC)  ▷ the replica creates two distinct...
                                                                 ▷ proposals
5:       v1 ← generateVoteMessages(p1, f)        ▷ then, the replica generates f VOTE messages for p1, and...
6:       v2 ← generateVoteMessages(p2, f)        ▷ f VOTE messages for p2, one for each Byzantine replica
7:       set1, set2 ← getTwoRandomSets(e, k, honestReplicas)   ▷ lastly, the replica divide honest replicas...
                                                                 ▷ in two random sets of size k
8:       send p1 and v1 to set1          ▷ then, it sends the first proposal and votes for it to the first set, set1,
9:       send p2 and v2 to set2          ▷ while, sending the second proposal and its votes to the second set, set2
10: Procedure LaunchAmnesiaAttack(                               ▷ ***AMNESIA-ATTACK***
11:  e, validBlock, honestReplicas, k, f)
12:    if isByzantineLeader(e) then                              ▷ if the epoch leader is a Byzantine replica:
13:      p ← generateAlternativeProposal(e, validBlock)          ▷ the replica generates...
                                                                 ▷ an alternative proposal for validBlock
14:      v ← generateVoteMessages(p, f)          ▷ then, the replica generates f VOTE messages for p,...
                                                                 ▷ one for each Byzantine replica
15:      send p and v to honestReplicas                          ▷ the replica sends the proposal and...
                                                                 ▷ votes for it to the all honest replicas
16:    else                                                      ▷ else, if the leader is an honest replica:
17:      set1, set2 ← getTwoRandomSets(e, k, honestReplicas)     ▷ the replica divides honest replicas...
                                                                 ▷ in two random sets of size k
18:      upon receiving p = ⟨PROPOSE, e, b, BC⟩l                 ▷ then, when it receives the proposal...
19:       where l = leader(e) do                                 ▷ from epoch leader...
20:         v ← generateVoteMessages(p, f)          ▷ it generates f VOTE messages for received proposal and...
21:         b ← generateBlameMessages(e, f)   ▷ f BLAME messages for epoch e, one for each Byzantine replica
22:         send v to set1                  ▷ then, it sends votes to one subset of honest replicas, set1,...
23:         send b to set2                ▷ while sending blames to the second subset of honest replicas, set2
```

All Byzantine replicas launch the same attack, not only the current epoch leader. This ensures that messages arrive at their destinations as fast as possible. So, if the malicious leader is far from some honest replica, the honest replica will receive attack messages from its closest Byzantine replica. For example, if the attack is *EQUIVOCATION-ATTACK*, each Byzantine replica will generate two proposals and votes for these proposals and send one proposal and its votes to one subset of honest replicas and another proposal and its votes to the other subset of honest replicas.

In attacks where Byzantine replicas divide honest replicas into two subsets and send different messages to them, the subsets are picked randomly. The size of these subsets is a parameter of the algorithm, defined by k. If k is set to 2, function $getTwoRandomSets(e, k, set)$ will return two different subsets, each containing two random elements. Byzantine replicas ensure they have the same subsets by using the current epoch number as a random number generator seed.

Algorithm 5 Byzantine attacks on BoundBFT's progress

```
1: Procedure LaunchBlameAttack(                                   ▷ ***BLAME-ATTACK***
2:  e, honestReplicas, f)
3:    if isHonestLeader(e) then                                   ▷ if the epoch leader is honest:
4:      b ← generateBlameMessages(e, f)                           ▷ the replica generates f BLAME messages,...
                                                                  ▷ one for each Byzantine replica, and...
5:      send b to honestReplicas                                  ▷ send them to all honest replicas
6: Procedure LaunchEquivocationCertAttack(                        ▷ ***EQUIVOCATION-CERT-ATTACK***
7:  e, validBlock, validBC, k, honestReplicas, f) :
8:    if isByzantineLeader(e) then                                ▷ if the epoch leader is Byzantine:
9:      p1, p2 ← generateTwoDifferentProposals(e, validBlock, validBC)   ▷ the replica generates...
                                                                  ▷ two different proposals, and...
10:     v1 ← generateVoteMessages(p1, f)        ▷ f VOTE messages only for p1, one for each Byzantine replica
11:     set1, set2 ← getTwoRandomSets(e, k, honestReplicas)       ▷ then, the replica divides honest replicas...
                                                                  ▷ in two random sets of size k
12:     send p1 and v1 to set1                                    ▷ finally, it sends the first proposal and votes for it...
                                                                  ▷ to the first set of honest replicas, set1,...
13:     send p1 and p2 to set2          ▷ while sending both proposals to the second set of honest replicas, set2
14: Procedure LaunchBlameCertAttack(                              ▷ ***BLAME-CERT-ATTACK***
15:  e, validBlock, validBC, k, honestReplicas, f) :
16:    if isByzantineLeader(e) then                                ▷ if the epoch leader is Byzantine:
17:     p ← generateNewProposal(e, validBlock)                    ▷ the replica generates new proposal for epoch e,...
18:     v ← generateVoteMessages(p, f)                            ▷ f VOTE messages for proposal, and...
19:     b ← generateBlameMessages(e, f)                           ▷ f BLAME messages, one for each Byzantine replica
20:     set1, set2 ← getTwoRandomSets(e, k, honestReplicas)       ▷ then, replica divides honest replicas...
                                                                  ▷ in two random sets of size k
21:     send p and v to set1                                      ▷ finally, it sends the proposal and votes for it...
                                                                  ▷ to the first set of honest replicas, set1,...
22:     send b to set2                  ▷ while sending blame messages to the second set of honest replicas, set2
```

B Determining BoundBFT's synchrony bound

This section presents the complete data of the experiments we used to determine BoundBFT's synchronous bound (see Section 5.2). Namely, we implemented all proposed attacks (see Algorithm 3 and Figures 4 and 5). Then, we ran BoundBFT in our cluster while varying the number of Byzantine replicas f. As a starting point, we set Δ to 1250 ms (99.99%), the synchronous bound from [30], and gradually decreased it to the point where we started to observe BoundBFT's agreement and progress violations. More than 300 hours worth of experiments were conducted in total. Tables 3 and 4 below show data for 1KB and 32KB block sizes, respectively.

N = 60, block size = 1KB												
Δ (ms)	**No attack**											
	Agreement						Progress					
1250	0%						0%					
600	0%						0%					
300	0%						0%					
150	0%						0%					
100	0%						0%					
50	0%						65%					
Δ (ms)	**Equivocation attack**											
	f=29				f=19				f=1			
	k_{min}		k_{max}		k_{min}		k_{max}		k_{min}		k_{max}	
	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
150	0%	1%	0%	3%	0%	0%	0%	0%	0%	0%	0%	0%
100	5%	36%	6%	6%	0%	5%	0%	0%	0%	0%	0%	0%
50	19%	80%	31%	60%	2%	73%	8%	54%	0%	65%	0%	65%
Δ (ms)	**Amnesia attack**											
	f=29				f=19				f=1			
	k_{min}		k_{max}		k_{min}		k_{max}		k_{min}		k_{max}	
	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
150	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
100	4%	19%	0%	18%	0%	14%	0%	8%	0%	0%	0%	0%
50	26%	90%	0%	80%	9%	81%	0%	97%	0%	66%	0%	67%
Δ (ms)	**Blame attack**											
	f=29				f=19				f=1			
	Agreement		Progress		Agreement		Progress		Agreement		Progress	
1250	0%		0%		0%		0%		0%		0%	
600	0%		0%		0%		0%		0%		0%	
300	0%		0%		0%		0%		0%		0%	
150	0%		0%		0%		0%		0%		0%	
100	0%		84%		0%		46%		0%		0%	
50	0%		100%		0%		100%		0%		67%	
Δ (ms)	**Equivocation certificate attack**											
	f=29				f=19				f=1			
	k_{min}		k_{max}		k_{min}		k_{max}		k_{min}		k_{max}	
	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
150	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
100	0%	16%	0%	12%	0%	17%	0%	12%	0%	0%	0%	0%
50	2%	88%	0%	90%	0%	86%	0%	80%	0%	63%	0%	66%
Δ (ms)	**Blame certificate attack**											
	f=29				f=19				f=1			
	k_{min}		k_{max}		k_{min}		k_{max}		k_{min}		k_{max}	
	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
150	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
100	0%	21%	0%	25%	0%	15%	0%	17%	0%	0%	0%	0%
50	2%	90%	0%	92%	0%	92%	0%	99%	0%	63%	0%	64%

Table 3: Percentage of Agreement and Progress violations when running BoundBFT under different attacks while using different values as its Δ. The table shows data for the setup of 60 replicas, 1KB block size and different number of Byzantine replicas (f). Additionally, for attacks that parition honest replicas in two subsets, it shows results when Byzantine replicas divide them in two minimal subsets (k=kmin=1) and two maximal subsets (k=kmax=n-f/2).

N = 60, block size = 32KB												
Δ (ms)	**No attack**											
	Agreement						Progress					
1250	0%						0%					
600	0%						0%					
300	0%						0%					
150	0%						0%					
100	0%						0%					
50	0%						66%					
Δ (ms)	**Equivocation attack**											
	f=29				f=19				f=1			
	k_{min}		k_{max}		k_{min}		k_{max}		k_{min}		k_{max}	
	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	5%	0%	0%	0%	0%	0%	0%	0%	0%
150	2%	0%	0%	15%	0%	0%	0%	0%	0%	0%	0%	0%
100	5%	34%	0%	67%	0%	13%	0%	9%	0%	0%	0%	0%
50	27%	91%	5%	85%	1%	94%	0%	97%	0%	66%	0%	66%
Δ (ms)	**Amnesia attack**											
	f=29				f=19				f=1			
	k_{min}		k_{max}		k_{min}		k_{max}		k_{min}		k_{max}	
	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	2%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
150	3%	26%	1%	10%	0%	1%	0%	0%	0%	0%	0%	0%
100	6%	42%	0%	44%	0%	22%	0%	0%	0%	0%	0%	0%
50	27%	81%	10%	88%	9%	74%	0%	37%	0%	67%	0%	67%
Δ (ms)	**Blame attack**											
	f=29				f=19				f=1			
	Agreement		Progress		Agreement		Progress		Agreement		Progress	
1250	0%		0%		0%		0%		0%		0%	
600	0%		0%		0%		0%		0%		0%	
300	0%		0%		0%		0%		0%		0%	
150	0%		25%		0%		0%		0%		0%	
100	0%		85%		0%		43%		0%		0%	
50	0%		100%		0%		99%		0%		79%	
Δ (ms)	**Equivocation certificate attack**											
	f=29				f=19				f=1			
	k_{min}		k_{max}		k_{min}		k_{max}		k_{min}		k_{max}	
	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	2%	0%	0%	0%	0%	0%	0%	0%	0%
150	0%	0%	0%	18%	0%	0%	0%	0%	0%	0%	0%	0%
100	0%	31%	0%	61%	0%	13%	0%	17%	0%	0%	0%	0%
50	3%	86%	0%	83%	0%	95%	0%	95%	0%	67%	0%	67%
Δ (ms)	**Blame certificate attack**											
	f=29				f=19				f=1			
	k_{min}		k_{max}		k_{min}		k_{max}		k_{min}		k_{max}	
	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress	Agreement	Progress
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
150	0%	2%	0%	3%	0%	0%	0%	0%	0%	0%	0%	0%
100	0%	30%	0%	32%	0%	17%	0%	12%	0%	0%	0%	0%
50	1%	90%	0%	96%	0%	95%	0%	99%	0%	66%	0%	66%

Table 4: Percentage of Agreement and Progress violations when running BoundBFT under different attacks while using different values as its Δ. The table shows data for the setup of 60 replicas, 32KB block size and different number of Byzantine replicas (f). Additionally, for attacks that parition honest replicas in two subsets, it shows results when Byzantine replicas divide them in two minimal subsets (k=kmin=1) and two maximal subsets (k=kmax=n-f/2).