Practical Synchronous Byzantine Consensus

Ling Ren MIT
Kartik Nayak UMD
Ittai Abraham VMware Research
Srinivas Devadas MIT

Abstract

We present new protocols for Byzantine state machine replication and Byzantine agreement in the synchronous and authenticated setting. The celebrated PBFT state machine replication protocol tolerates f Byzantine faults in an asynchronous setting using $3f + 1$ replicas, and has since been studied or deployed by numerous works. In this work, we improve the Byzantine fault tolerance to $n = 2f + 1$ by utilizing the synchrony assumption. The key challenge is to ensure a quorum intersection at one *honest* replica. Our solution is to rely on the synchrony assumption to form a *post-commit* quorum of size $2f + 1$, which intersects at $f + 1$ replicas with any *pre-commit* quorums of size $f + 1$. Our protocol also solves synchronous authenticated Byzantine agreement in fewer rounds than the best existing solution (Katz and Koo, 2006). A challenge in this direction is to handle non-simultaneous termination, which we solve by introducing a notion of *virtual* participation after termination. Our protocols may be applied to build practical synchronous Byzantine fault tolerant systems and improve cryptographic protocols such as secure multiparty computation and cryptocurrencies when synchrony can be assumed.

1 Introduction

Byzantine consensus [24, 7] is a fundamental problem in distributed computing and cryptography. It has been used to build fault tolerant systems such as distributed storage systems [34, 8, 3, 21, 1, 9, 20], certificate authorities [33, 38], fair peer-to-peer sharing [36], and more recently cryptocurrencies [18, 28, 30, 2]. It has also been frequently used as building blocks in cryptographic protocols such as secure multi-party computation [16, 5].

Broadly speaking, Byzantine consensus considers the problem of reaching agreement among a group of n parties, among which up to f can have Byzantine faults and deviate from the protocol arbitrarily. There exist a few variant formulations for the Byzantine consensus problem.[1] Two theoretical formulations are Byzantine broadcast and Byzantine agreement [31, 24]. In Byzantine broadcast, there is a designated *sender* who tries to broadcast a value to the parties; In Byzantine agreement, every party holds an initial input value. To rule out trivial solutions, both problems have additional validity requirements. Byzantine broadcast and agreement have been studied under various combinations of timing (synchrony, asynchrony or partial synchrony) and cryptographic assumptions (whether or not to assume digital signatures). It is now well understood that these assumptions drastically affect the bounds on fault tolerance and solutions to the problems. In particular, Byzantine agreement requires $f < n/3$ under partial synchrony or asynchrony even with digital signatures but can be achieved with $f < n/2$ under synchrony.

Most Byzantine broadcast and agreement protocols have been designed to demonstrate theoretical feasibility and the problem definitions are also not always convenient to work with in practice. A more practice-oriented problem formulation is Byzantine fault tolerant (BFT) state machine replication [35, 7]. In this formulation, the goal is to design a replicated service that provides the same interface as a single server, despite some replicas experiencing Byzantine faults. In particular, honest replicas agree on a sequence of values and their *order*, while the validity of the values is left outside the protocol. The PBFT protocol by Castro and Liskov [7] is an asynchronous state machine replication protocol that achieves the optimal

[1]We use the word "consensus" as a collective term for these variants; other papers have different conventions.

$f < n/3$ Byzantine fault tolerance. As the first BFT protocol designed for practical efficiency, PBFT has since inspired numerous follow-up works including many practical systems [34, 8, 3, 21, 38, 36, 37, 1, 9, 27, 19].

Perhaps somewhat surprisingly, we do not yet have a practical solution for Byzantine consensus in the seemingly easier synchronous and authenticated (i.e., with digital signatures) setting. To the best of our knowledge, the most efficient Byzantine agreement protocol with the optimal $f < n/2$ fault tolerance in this setting is due to Katz and Koo [17], which requires in expectation 24 rounds of communication (not counting the random leader election subroutine). To agree on many messages sequentially, it requires additional generic transformations [25, 17] that further increase the expected round complexity to a staggering 72 rounds per instance!

This paper presents efficient Byzantine consensus protocols for the synchronous and authenticated setting tolerating $f < n/2$ faults. Our main focus is BFT state machine replication, for which our protocol requires amortized 4 to 8 rounds per slot (we say each value in the sequence fills one *slot*). In scenarios where synchrony can be assumed, we believe our protocol can be applied to build practical Byzantine fault tolerant systems and services tolerating $f < n/2$ Byzantine faults. Meanwhile, our protocol can also solve multi-valued Byzantine broadcast and agreement for $f < n/2$ in expected 10 rounds assuming a random leader oracle. (The increase to 10 rounds is due to a technicality in problem formulation. Our protocol barely changes.)

1.1 Overview of Our Protocol

Interestingly, our protocol draws inspiration from the Paxos protocol [23], which is neither synchronous nor Byzantine fault tolerant. The core of our protocol resembles the synod algorithm in Paxos, adapted to the synchronous and Byzantine setting while maintaining the $f < n/2$ fault tolerance. In this sense, another appropriate name for our protocol is perhaps "synchronous Byzantine Paxos". Since our main focus is state machine replication, we will describe the core protocol with "replicas" instead of "parties".

In a nutshell, our synchronous Byzantine synod algorithm runs in iterations with a unique leader in each iteration (how to elect leaders is left to higher level protocols). Each new leader picks up the states left by previous leaders and drives agreement in its iteration. A Byzantine leader can prevent progress but cannot violate safety. As soon as an honest leader emerges, then all honest replicas reach agreement and terminate at the end of that iteration.

While synchrony is supposed to make the problem easier, it turns out to be non-trivial to adapt the synod algorithm to the synchronous and Byzantine setting while achieving the optimal $f < n/2$ fault tolerance. The major challenge is to ensure *quorum* intersection [23] at one *honest* replica. The core idea of the synod algorithm in Paxos is to form a quorum of size $f + 1$ before a commit. $n = 2f + 1$ guarantees that two quorums always intersect at one replica. This replica in the intersection, which is honest in Paxos, will force a future leader to respect the committed value. In order to tolerate f Byzantine faults, PBFT uses quorums of size $2f + 1$ out of $n = 3f + 1$, so that two quorums intersect at $f + 1$ replicas, among which one is guaranteed to be honest. At first glance, our goal of one honest intersection seems implausible with the $n = 2f + 1$ constraint. Following PBFT, we need two quorums to intersect at $f + 1$ replicas which seems to require quorums of size $1.5f + 1$. On the other hand, a quorum size larger than $f + 1$ (the number of honest replicas) seems to require participation from Byzantine replicas and thus lose liveness. Our solution is to utilize the synchrony assumption to form a *post-commit quorum* of size $2f + 1$. A post-commit quorum does not affect liveness and intersects with any *pre-commit quorum* (of size $f + 1$) at $f + 1$ replicas, satisfying the requirement of one honest replica in intersection.

Another challenge arises when we apply our core protocol to Byzantine broadcast and agreement, that is, how to ensure termination for all honest parties. In general, it is very easy for faulty parties to make some honest parties commit while preventing other honest parties from committing. If a single honest party terminates earlier than others, then all honest parties that have not terminated will never be able to terminate (unable to form a quorum of size $f + 1$) if Byzantine parties simply stop participating. This is not a problem for state machine replication protocols as they are designed to run forever, but it is vital for a Byzantine broadcast or agreement instance in isolation. To solve this problem, we introduce techniques to let honest parties participate "virtually" even after termination. This way, once there is an honest leader, it will bring all honest parties to termination.

With some additional checks and optimizations, we obtain our core protocol: a 4-round synchronous Byzantine synod protocol (three Paxos-like rounds plus a notification round). It preserves safety under Byzantine leaders and ensures termination once an honest leader emerges. It is then straightforward to apply this core protocol to state machine replication as well as Byzantine broadcast and agreement in the synchronous and authenticated setting with $f < n/2$. For state machine replication, we simply rotate the leader role among the replicas after each iteration. Because each honest leader is able to fill at least one slot, the protocol spends amortized 2 iterations (8 rounds) per slot in the presence of $f = \lfloor \frac{n-1}{2} \rfloor$ faults. In the best case where there is no fault, the protocol fills one slot in every iteration (4 rounds). For Byzantine broadcast and agreement, assuming a random leader election subroutine (and not counting its cost), our solution runs in expected 10 rounds (expected 3 iterations with the first and the last iteration skipping a round). We remark that our protocol achieves the optimal $f < n/2$ fault tolerance for synchronous authenticated Byzantine agreement and BFT state machine replication, but not for Byzantine broadcast. Our quorum-based approach cannot solve synchronous authenticated Byzantine broadcast in the dishonest majority case ($f \geq n/2$).

2 Related Work

Byzantine agreement and broadcast. The Byzantine agreement and Byzantine broadcast problems were first introduced by Lamport, Shostak and Pease [24, 31]. They presented protocols and fault tolerance bounds for two settings (both synchronous). Without cryptographic assumptions (the unauthenticated setting), Byzantine broadcast and agreement can be solved if $f < n/3$. Assuming digital signatures (the authenticated setting), Byzantine broadcast can be solved if $f < n$ and Byzantine agreement can be solved if $f < n/2$. The initial protocols had exponential message complexities [31, 24]. Fully polynomial protocols were later shown for both the authenticated ($f < n/2$) [10] and the unauthenticated ($f < n/3$) [15] settings. Both protocols require $f + 1$ rounds of communication, which matches the lower bound on round complexity for deterministic protocols [13]. To circumvent the $f + 1$ round lower bound, a line of work explored the use of randomization [4, 32] which eventually led to expected constant-round protocols for both the authenticated ($f < n/2$) [17] and the unauthenticated ($f < n/3$) [12] settings. In the asynchronous setting, the FLP impossibility [14] rules out any deterministic solution. Some works use randomization [4, 6] or partial synchrony [11] to circumvent the impossibility.

State machine replication. A more practical line of work studies state machine replication [22, 35]. The goal is to design a distributed system consisting of replicas to process requests from external clients while behaving like a single-server system. Paxos [23] and Viewstamped replication [29] tolerate f crash faults with $n \geq 2f + 1$ replicas. The PBFT protocol [7] tolerates f Byzantine faults with $n \geq 3f + 1$ replicas. In all three protocols, safety is preserved even under asynchrony while progress is made only during synchronous periods. Numerous works have extended, improved or deployed PBFT [34, 8, 3, 21, 38, 36, 37, 1, 9, 27, 19]. They all consider the asynchronous setting and require $n \geq 3f + 1$.

To the best of our knowledge, the only work on state machine replication that considers the exact same setting as ours (Byzantine faults, synchrony, digital signatures and $n \geq 2f + 1$) is XFT [26]. It relies on an active group of $f + 1$ honest replicas to make progress. XFT is designed to optimize efficiency for small n and f (e.g., $f = 1$). Its performance degrades rapidly as n and f increase, especially when there are actually $f = \lfloor \frac{n-1}{2} \lfloor$ faults. In that case, among the $\binom{n}{f+1}$ $f + 1$-sized groups in total, only one is all-honest, and XFT would require an exponential number of view changes to find that group. In contrast, our protocol requires at most amortized 2 iterations per slot independent of n and f.

3 A Synchronous Byzantine Synod Protocol

3.1 Model and Overview

Our core protocol is a synchronous Byzantine synod protocol. We have $n = 2f + 1$ replicas $1, 2, \cdots n$. Up to f of them may be Byzantine and may deviate from the protocol arbitrarily. The core protocol's goal is to guarantee that all honest replicas eventually commit (liveness) and commit on the same value (safety).

We assume synchrony. If an honest replica i sends a message to another honest replica j at the beginning of a round, the message is guaranteed to reach by the end of that round. Our protocol runs in iterations and each iteration consists of 4 rounds. We describe the protocol assuming all replicas have perfectly synchronized clocks. Hence, they enter each round simultaneously and have the same view on the current iteration number k (the first iteration has $k = 1$). In practice, it is sufficient to have a known bound on the communication delay.

We assume public key cryptography. Every replica knows the public (verification) key of every other replica, and they use digital signatures when communicating with each other. Byzantine replicas cannot forge honest replicas' signatures, which means messages in our systems enjoy authenticity as well as non-repudiation. We use $\langle x\rangle_i$ to denote a message x that is signed by replica i, i.e., $\langle x\rangle_i = (x, \sigma)$ where $\sigma = \mathsf{Sign}_i(x)$ is a signature produced by replica i using its private signing key. For better efficiency, σ can be a signature of a message's digest, i.e., output of a collision resistant hash function. A message can be signed by multiple replicas (or the same replica) in layers, i.e., $\langle\langle x\rangle_i\rangle_j = \langle x, \sigma_i\rangle_j = (x, \sigma_i, \sigma_j)$ where $\sigma_i = \mathsf{Sign}_i(x)$ and $\sigma_j = \mathsf{Sign}_j(x \mid\mid \sigma_i)$ ($\mid\mid$ denotes concatenation).

The core protocol assumes a unique leader in each iteration that is known to every replica. An iteration leader can be one of the replicas but can also be an external entity. Similar to Paxos, we decouple leader election from the core protocol and leave it to higher level protocols (Section 4 and 5) or, in some cases, the application level. For example, a cryptocurrency (blockchain) may elect leaders based on proof of work.

Each iteration consists of 4 rounds. The first three rounds are conceptually similar to Paxos: (1) the leader learns the states of the system, (2) the leader proposes a safe value, and (3) every replica sends a commit request to every other replica. If a replica receives $f + 1$ commit requests for the same value, it commits on that value. If a replica commits, it notifies all other replicas about the commit using a 4th round. Upon receiving a notification, other replicas *accept* the committed value and will vouch for that value to future leaders. To tolerate Byzantine faults, we need to add equivocation checks and other proofs of honest behaviors at various steps. We now describe the protocol in detail.

3.2 Detailed Protocol

Each replica i internally maintains some long-term states (v_i, k_i, C_i) across iterations to record its *accepted* value. Initially, each replica i initializes $(v_i, k_i, C_i) := (\perp, 0, \perp)$. If replica i later accepts a value v in iteration k, it sets $(v_i, k_i, C_i) := (v, k, C)$ such that C *certifies* that v is legally accepted in iteration k (see Table 1). In the protocol, honest replicas will only react to *valid* messages. Invalid messages are simply discarded. To keep the presentation simple, we defer the validity definitions of all types of messages to Table 1. We first describe the protocol assuming no replica has terminated, and later amend the protocol to deal with non-simultaneous termination.

Round 0 (status) Each replica i sends a $\langle\langle k, \mathsf{status}, v_i, k_i\rangle_i, C_i\rangle_i$ message to the leader L_k of the current iteration k, informing L_k of its current accepted value. We henceforth write L_k as L for simplicity.

At the end of this round, the leader L picks $f + 1$ valid status messages to form a *safe value proof* P.

Round 1 (propose) The leader L picks a value v that is *safe to propose* with respect to P: v should match the value that is accepted in the most recent iteration in P, or any v is safe to propose if no value has been accepted in P (see Table 1 for more details). L then sends a signed proposal $\langle\langle k, \mathsf{propose}, v\rangle_L, P\rangle_L$ to all replicas including itself.

At the end of this round, if replica i receives a valid proposal $\langle\langle k, \mathsf{propose}, v\rangle_L, P\rangle_L$ from the leader, it sets $v_{L\to i} := v$. Otherwise (leader is faulty), it sets $v_{L\to i} := \perp$.

Round 2 (commit) If $v := v_{L\to i} \neq \perp$, replica i forwards the proposal $\langle k, \mathsf{propose}, v\rangle_L$ (excluding P) and sends a $\langle k, \mathsf{commit}, v\rangle_i$ request to all replicas including itself. Otherwise, it sends nothing.

At the end of this round, if replica i is forwarded a valid proposal $\langle k, \mathsf{propose}, v'\rangle_L$ in which $v' \neq v_{L\to i}$, it does not commit in this iteration (leader has equivocated). Else, if replica i receives $f + 1$ valid

$\langle k, \mathsf{commit}, v\rangle_j$ requests in all of which $v = v_{L\to i}$, it commits on v and sets its long-term state C_i to be these $f+1$ commit requests concatenated. In other words, replica i commits if and only if it receives $f+1$ valid and matching commit requests and does not detect leader equivocation.

Round 3 (notify) If replica i has committed on v at the end of Round 2, it sends a notification $\langle\langle k, \mathsf{notify}, v\rangle_i, C_i\rangle_i$ to every other replica, and terminates.

At the end of this round, if replica i receives a valid $\langle\langle k, \mathsf{notify}, v\rangle_j, C\rangle_j$ message, it accepts v by setting its long-term states $(v_i, k_i) := (v, k)$ and $C_i := C$. If replica i receives multiple valid notify messages with different values, it is free to accept any one or none. Lastly, replica i increments the iteration counter k and enters the next iteration.

Summaries and certificates. In a $\langle\langle k, \mathsf{status}, v_i, k_i\rangle_i, C_i\rangle_i$ message, we call the $\langle k, \mathsf{status}, v_i, k_i\rangle_i$ component a status summary, and the C_i component a *certificate*. Similarly, in a $\langle\langle k, \mathsf{notify}, v\rangle_i, C_i\rangle_i$ message, we call the $\langle k, \mathsf{notify}, v\rangle_i$ component a notify summary, and the C_i component a certificate. This distinction will be important soon for handling non-simultaneous termination.

A shorter safe value proof P. In the above basic protocol, P consists of $f+1$ valid status messages, each of which contains a certificate. This yields a length of $|P| = O(n^2)$. We observe that P can be optimized to have length $O(n)$. P can consist of $f+1$ valid status summaries plus a valid certificate for a summary that claims the highest iteration number. Table 1 presents more details.

Non-simultaneous termination. We need to ensure that all honest replicas eventually commit and terminate. However, in the protocol above, it is possible that some honest replicas terminate in an iteration while other honest replicas enter the next iteration. Without special treatment, the honest replicas who enter the new iteration will never be able to terminate (unable to gather $f+1$ matching commit requests) if Byzantine replicas simply stop participating.

To solve this problem, we need honest replicas to continue participating "virtually" even after termination. If replica t has committed on v_t and terminated in iteration k_t, it is supposed to send a valid notification $\langle\langle k_t, \mathsf{notify}, v_t\rangle_t, C_t\rangle_t$ to all other replicas. Upon receiving this notification, replica i becomes aware that replica t has terminated, and does not expect any messages from replica t in future iterations. Of course, it is also possible that replica t is Byzantine, and sends notifications only to a subset of replicas. We need to ensure that such a fake termination does not violate safety or liveness.

Assuming that replica i has received a valid notification $\langle\langle k_t, \mathsf{notify}, v_t\rangle_t, C_t\rangle_t$ from replica t, we now amend the protocol such that in all future iterations $k > k_t$, replica i "pretends" that it keeps receiving *virtual* messages from replica t in the follow ways:

- In **Round 0**, if replica i is the current iteration leader, it treats the notification $\langle\langle k_t, \mathsf{notify}, v_t\rangle_t, C_t\rangle_t$ as a valid $\langle\langle k, \mathsf{status}, v_t, k_t\rangle_t, C_t\rangle_t$ message from replica t. In particular, the safe value proof P is allowed to include the notify summary $\langle k_t, \mathsf{notify}, v_t\rangle_t$ in place of a status summary.

- In **Round 1**, if replica t is the current iteration leader L, then replica i treats the notify summary $\langle k_t, \mathsf{notify}, v_t\rangle_t$ as a virtual proposal for v_t. If $v_t = v_i$ (the value replica i accepts), then replica i considers the virtual proposal valid sets $v_{L\to i} = v_t$. Later in Round 2, replica i forwards the virtual proposal to all replicas for equivocation checking.

- In **Round 2**, replica i treats the notify summary $\langle k_t, \mathsf{notify}, v_t\rangle_t$ as a valid commit request for v_t from replica t. In particular, a certificate C_i is allowed to include this notify summary in place of a commit request.

Table 1: **Validity requirements of messages.** Every message includes an iteration number k and is signed by the sender. Some messages contain components signed by the sender or other replicas in layers. In order to be valid, a message must carry the correct current iteration number, and all signatures in the message must verify correctly. Additional validity requirements for every type of message are listed below. Note that the validity of a message might depend on the validity of its components and/or other conditions defined in the table.

Message	Validity requirements
$\langle\langle k, \mathsf{status}, v_i, k_i\rangle_i, C_i\rangle_i$	The initial states $(v_i, k_i, C_i) = (\perp, 0, \perp)$ is valid. If $k_i > 0$, then C_i must certify (v_i, k_i).
$\langle\langle k, \mathsf{propose}, v\rangle_L, P\rangle_L$	v is safe to propose with respect to P. Replica L is the leader for iteration k.
$\langle k, \mathsf{commit}, v\rangle_i$	No extra requirement.
$\langle\langle k, \mathsf{notify}, v\rangle_i, C\rangle_i$	C certifies (v, k).
v is safe to propose with respect to P	P consists of $f + 1$ status or notify summaries, plus a valid certificate for a summary that claims the highest non-zero iteration number. Each status summary in P must carry the current iteration number k, but a notify summary may not. If multiple summaries claim the same highest iteration number, the certificate can be for any of them. Without loss of generality, suppose P contains summaries from replica 1 to $f + 1$, i.e., $P = (s_1, s_2, \cdots, s_{f+1}, C)$ in which $s_j = \langle k, \mathsf{status}, v_j, k_j\rangle_j$ or $s_j = \langle k_j, \mathsf{notify}, v_j\rangle$. Let $k^* = \max(k_1, k_2, \cdots, k_{f+1})$. If $k^* = 0$, which implies $\forall j, (v_j, k_j) = (\perp, 0)$, then $C = \perp$ and any v is safe to propose with respect to P. If $k^* > 0$, then v must match some v_j such that $k_j = k^*$ and C certifies (v_j, k_j). There may be additional requirements on v at the application level that the replicas should check for.
C certifies (v, k)	A valid certificate C for (v, k) consists of $f + 1$ commit requests or notify summaries for value v. Each commit request in C must carry the current iteration number k, but a notify summary may not.

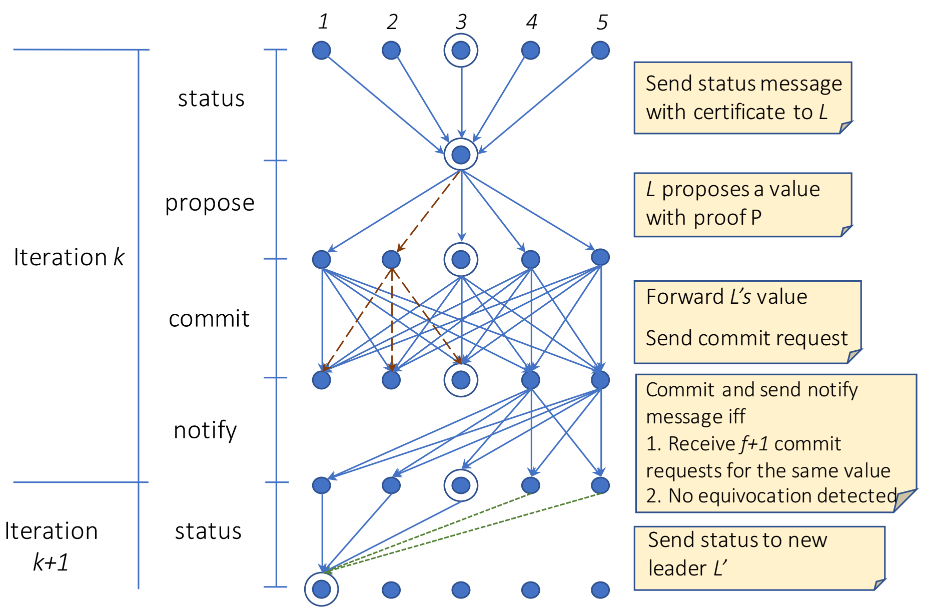
1
2
3
4
5
Iteration k
Iteration k+1
status
propose
commit
notify
status
Send status message with certificate to L
L proposes a value with proof P
Forward L's value
Send commit request
Commit and send notify message iff
1. Receive f+1 commit requests for the same value
2. No equivocation detected
Send status to new leader L'

Figure 1: **Synchronous Byzantine Synod Protocol with** $n = 2f + 1$. The figure shows an example for iteration k. In the example, replicas 2 and 3 are Byzantine. Also, replica 3 is the leader L in this iteration. **0.** (status) At the start of the iteration, each replica sends its current states to the leader L. In this example, no replica has committed or accepted any value, so L can propose any value of its choice. **1.** (propose) L equivocates and sends one proposal to replica 2 (shown by dashed red arrow) and a different proposal to honest replicas. **2.** (commit) Honest replicas forward L's proposal to all replicas, and send commit requests to all replicas. Replica 2 only sends to replicas $\{1, 2, 3\}$. **3.** (notify) Replicas 4 and 5 do not detect equivocation and receive $f + 1$ commit requests for the same value. They commit, notify all other replicas and terminate. Replica 1 detects leader equivocation and does not commit despite also receiving $f + 1$ commit requests for the blue value. **4.** (status) On receiving a valid notification, replica 1 accepts the blue value. The replicas send status messages to the new leader L' for iteration $k + 1$. status messages from terminated replicas 4 and 5 are virtual, i.e., their notify messages from iteration k are treated as status messages in iteration $k + 1$ (shown by dotted green arrows).

3.3 Safety and Liveness

In this section, we prove that the protocol in Section 3.2 provides safety and liveness. We will also give intuition to aid understanding.

The scenario to consider for safety is when an honest replica h commits on a value v^* in iteration k^*. We show that, in all subsequent iterations, no leader can construct a valid proposal for a value other than v^*. We first show that Byzantine replicas cannot commit or accept a value other than v^* in iteration k^*. Thus, all other honest replicas accept v^* at the end of iteration k^* upon receiving notify from the honest replica h. The leader in iteration $k^* + 1$ needs to show $f + 1$ status messages and pick a value with the highest iteration number (cf. Table 1). One of these status messages must be from an honest replica and contain v^*. This implies that a value other than v^* cannot be proposed in iteration $k^* + 1$, and hence cannot be committed or accepted in iteration $k^* + 1$, and hence cannot be proposed in iteration $k^* + 2$, and so on. Safety then holds by induction.

We now formalize the above intuition through an analysis on certificates. A certificate C certifies that v has been legally committed and/or accepted in iteration k, if it meets the validity requirement in Table 1. We prove the following lemma about certificates: once an honest replica commits, all certificates in that iteration and future iterations can only certify its committed value.

Lemma 1. *Suppose replica h is the first honest replica to commit and terminate. If replica h commits on v^* in iteration k^* and C certifies (v, k) where $k \geq k^*$, then $v = v^*$.*

Proof. We prove by induction on k. For the base case, suppose C certifies (v, k^*). C must consist of $f+1$ valid commit requests or notify summaries for v. At least one of these is from an honest replica (call it h_1). Since no honest replica has terminated so far, replica h_1 must have sent a normal commit request rather than a virtual one. Thus, replica h_1 must have received a valid proposal (could be virtual) for v from the leader, and must have forwarded the proposal to all other replicas. If $v \neq v^*$, replica h would have detected leader equivocation, and would not have committed on v^* in this iteration. So we have $v = v^*$.

Before proceeding to the inductive case, it is important to observe that all honest replicas will accept v^* at the end of iteration k^*. This is because in iteration k^*, the honest replica h must have sent a notify message (with a certificate for v^*) to all replicas and no certificates for other values can exist. Now for the inductive case, suppose the lemma holds up to iteration k. We need to prove that if C certifies $(v, k+1)$, then $v = v^*$. The inductive hypothesis says between iteration k^* to iteration k, all valid certificates certify v^*. So at the beginning of iteration $k+1$, all honest replicas either have committed on v^*, or still accept v^*. If C certifies $(v, k+1)$, it must consist of $f+1$ valid commit requests or notify summaries for v. At least one of these is from an honest replica (call it h_2).

1. If replica h_2 has terminated before iteration $k+1$, its notify summary (virtual commit request) is for v^* and we have $v = v^*$.
2. Otherwise, replica h_2 must have received from the leader a valid proposal (could be virtual) for v in iteration $k+1$. Note again that all honest replicas either have committed on v^* or still accept v^* at the beginning of iteration $k+1$.
   (a) If the proposal is a virtual one, in order for replica h_2 to consider it valid, v must match v_{h_2} (the value replica h_2 accepts), which is v^*.
   (b) If the proposal is a normal one, then it must contain a safe value proof P for v. P must include at least one honest replica's status or notify summary for (v^*, k^*) (or an even higher iteration number). Due to the inductive hypothesis, P cannot contain a valid certificate for (v', k') where $k' \geq k^*$ and $v' \neq v^*$. Therefore, the only value that is safe to propose with respect to P is $v = v^*$.

□

Theorem 1 (Safety)**.** *If two honest replicas commit on v and v' respectively, then $v = v'$.*

Proof. Suppose replica h is the first honest replica to commit, and it commits on v^* in iteration k^*. In order for another honest replica to commit on v, there must be a valid certificate C for (v, k) where $k \geq k^*$. Due to Lemma 1, $v = v^*$. Similarly, $v' = v^*$, and we have $v = v'$. □

Now we move on to liveness and show that an honest leader will guarantee that all honest replicas terminate by the end of that iteration.

Theorem 2 (Liveness)**.** *If the leader L in iteration k is honest, then every honest replica terminates at the end of iteration k (if it has not already terminated before iteration k).*

Proof. The honest leader L will send a proposal (could be virtual) to all replicas. If L has not terminated, it will send a valid proposal for a value v that is safe to propose, and attach a valid proof P. If L has committed on v and terminated, then all honest replicas have either committed on v or accept v at the beginning of iteration k (see proof of Lemma 1), so they all consider L's virtual proposal valid. Additionally, the unforgeability of digital signatures prevents Byzantine replicas from falsely accusing L of equivocating. Therefore, all honest replicas (terminated or otherwise) will send commit requests (could be virtual) for v, receive $f+1$ commit requests for v and terminate at the end of the iteration. □

Finally, we mention an interesting scenario that does not have to be explicitly addressed in the proofs. Before any honest replica commits, Byzantine replicas may obtain certificates for multiple different values in the same iteration. In particular, the Byzantine leader proposes two values v and v' to all the f Byzantine replicas. (An example with more than two values is similar.) Byzantine replicas then exchange f commit requests for both values among them. Additionally, the Byzantine leader proposes v and v' to different honest replicas. Now with one more commit request for each value from honest replicas, Byzantine replicas can obtain certificates for both v and v', and can make honest replicas accept different values by showing them different certificates (notify messages). However, this will not lead to a safety violation because no honest replica would have committed in this iteration: the leader has equivocated to honest replicas, so all honest replicas will detect equivocation from forwarded proposals and thus refuse to commit. This scenario showcases the necessity of both the synchrony assumption and the use of digital signatures for our protocol. Lacking either one, equivocation cannot be reliably detected and any protocol will be subject to the $f < n/3$ bound. For completeness, we note that the above scenario will not lead to a liveness violation, either. In the next iteration, honest replicas consider a proposal for any value (including v and v') to be valid as long as it contains a valid safe value proof P for that value.

4 Byzantine Fault Tolerant State Machine Replication

4.1 Model and Overview

The state machine replication approach for fault tolerance considers a scenario where clients submit requests to a replicated service [22, 35]. The replicated service should provide safety and liveness even when some replicas are faulty ($f < n/2$ Byzantine faults in our case). Safety means the service behaves like a single non-faulty server, and liveness means the service keeps processing client requests and eventually commits every request [35, 7].

We remark that BFT state machine replication requires an honest majority (i.e., $n \geq 2f + 1$) even in the synchronous setting [35]. Otherwise, Byzantine replicas in the majority can convince clients of a decision and later deny having ever committed that decision. This is in contrast to Byzantine broadcast which can be solved even for $n/2 \leq f \leq n - 2$. In Byzantine broadcast, honest parties just need to stay in agreement with each other and do not have to convince external entities (e.g., clients) of the correct system states. This distinction between BFT state machine replication and Byzantine broadcast becomes unimportant in the asynchronous setting in which both problems require $n \geq 3f + 1$.

To satisfy safety, honest replicas should agree on a sequence of values and their order. We say each value in the sequence occupies a *slot*. Our state machine replication protocol essentially runs a series of synod instances sequentially, one instance per slot. However, challenges arises as replicas can now be working on different slots in one iteration. This section discusses how to extend our core protocol to handle multiple slots.

We assume a replica does not commit slots out of order, i.e., it will only commit a value in slot s if it has committed in every slot numbered lower than s. Sequential commit is required in many scenarios where the validity of a value depends on values committed into previous slots. If sequential commit is not required, our protocol can be modified to run multiple slots independently in parallel.

4.2 The Protocol

First, the following changes are natural:

1. Each replica i internally maintains an additional long-term state s_i to denote which slot it is currently working on. $s_i = s$ means replica i has committed for slots $\{1, 2, \cdots, s-1\}$, and has not committed for slots $s, s+1, s+2, \cdots$.

2. All messages in Section 3.2 and Table 1 contain an additional slot number s. The four types of messages now have the form: $\langle\langle s, k, \mathsf{status}, v_i, k_i\rangle_i, C_i\rangle_i$, $\langle\langle s, k, \mathsf{propose}, v\rangle_L, P\rangle_L$, $\langle s, k, \mathsf{commit}, v\rangle_i$, and $\langle\langle s, k, \mathsf{notify}, v\rangle_i, C\rangle_i$. A certificate C now certifies a triplet (s, v, k) and must contain $f + 1$ commit

requests (could be virtual) for the same slot s. Similarly, a safe value proof P must now contain $f+1$ status summaries (could be virtual) for the same slot s.

Obviously, replica i follows the protocol from Section 3.2 to send and react to messages for its current slot s_i. If replica i receives messages for a past slot $s < s_i$, it can safely ignore these messages because its virtual messages (notify messages) for slot s will suffice to help any honest replicas terminate for that slot. The more interesting question is how to react to messages for future slots. We observe that it is vital that all honest replicas accept values for future slots upon receiving valid notify messages. If the sender of the notify message is an honest replica, then all other honest replicas must accept that value to prevent any other value from being proposed for that slot. However, apart from notify messages, honest replicas can ignore all other messages for future slots without violating safety. The reason is that the proof for safety from Section 3.3 only relies on honest replicas *not sending improper messages.* This implies the following changes:

3. The accepted states of a replica i are now per slot: $\mathsf{state}_i[s] = (v_i[s], k_i[s], C_i[s])$. Upon receiving a valid notify message $\langle\langle s, k, \mathsf{notify}, v\rangle_j, C\rangle_j$ for a future slot $s > s_i$, replica i sets $\mathsf{state}_i[s] := (v, k, C)$. Replica i ignores messages other than notify for future slots.

Safety is not affected as mentioned earlier. We next analyze liveness and the amortized round complexity for our state machine replication protocol. Note that now an honest leader may not be able to ensure termination for the slot it is working on because some honest replicas may be lagging behind. They may not react to its proposal; in fact, the leader may not even gather enough status messages to construct a valid proposal. However, each honest leader can still guarantee termination for at least one slot: the lowest numbered slot s^* that any honest replica is working on. After this iteration, all honest replicas will at least be working on slot s^*+1. Therefore, we simply rotate the leader in a round robin fashion, i.e., the leader for iteration k is replica $(k \mod n)$. This way, we can fill at least $f+1$ slots in a full rotation of $2f+1$ iterations, thereby achieving amortized 2 iterations (8 rounds) per slot. In the best case where there is no fault, the protocol spends 1 iteration (4 rounds) per slot.

Remark: In our protocol, a single replica's notify message is insufficient to convince a client that a value is committed. The reason is that a Byzantine replica may obtain a certificate for a value (and hence a valid notify message for it) but not send it to all other replicas, in which case a different value may be committed later. Therefore, a client needs to see $f+1$ notify messages (summaries suffice) from distinct replicas to be confident of a committed value. When n is large, it may be costly for a client to maintain connections to all n replicas. An alternative solution is to let a replica gather $f+1$ notify summaries at the end of Round 3, and send them to a client in a single message. This way, a client can be assured of a committed value by learning from a single replica at the cost of one extra round of latency.

5 Byzantine Broadcast and Agreement

In Byzantine broadcast, there is a designated *sender* who tries to broadcast a value to n parties. A solution needs to satisfy three requirements:

(termination) all honest parties eventually commit,

(agreement) all honest parties commit on the same value, and

(validity) if the sender is honest, then all honest parties commit on the value it broadcasts.

Our core protocol in Section 3.2 can be used to solve synchronous authenticated Byzantine broadcast for the $f < n/2$ case. We let the designated sender be the leader for the first iteration. Note that the status round can be skipped in the first iteration, and the sender can propose any value without attaching a safe value proof P. After the first iteration, we rotate the leader role among all n parties. It is straightforward to see that this solution achieves both agreement and validity. If the designated sender is honest, every honest party agrees on its value and terminates in the first iteration. Otherwise, the first honest leader that appears down the line will ensure agreement and termination for all honest parties. Assuming we have a random leader oracle, there is a $(f+1)/(2f+1) > 1/2$ probability that each leader after the first iteration is honest, so the protocol terminates in expected 2 iterations after the first iteration. Since the first iteration skips the status round, and honest replicas terminate without waiting for the notify messages in the last iteration, the protocol terminates in 3 iterations and 10 rounds in expectation.

The remaining question is how to elect random leaders. For this step, we can adopt the moderated verifiable secret sharing approach from Katz and Koo [17] [2] or the unique signature approach from Algorand [28]. The resulting protocol does not yet achieve expected constant rounds against an adaptive adversary who can corrupt a leader as soon as it is elected. Inspired by prior work [17, 28], the solution is to elect a random leader only *after* it has fulfilled the leader's responsibility. Adapting the idea to our protocol, every party should act as a leader in the status round and the propose round of each iteration to collect states and make a proposal. Then, a random leader is elected after the propose round and before the commit round. This way, there is a 1/2 chance of having an honest leader in each iteration even against an adaptive adversary.

In Byzantine agreement, every party holds an initial input value. A solution needs to satisfy the same termination and agreement requirements as in Byzantine broadcast. There exist a few different validity notions. We adopt a common one called strong unanimity [11]:

(validity) if all honest parties hold the same initial input value, then they all commit on that value.

To solve Byzantine agreement, we can use a classical transformation [31, 24]. Each party initiates a Byzantine broadcast in parallel to broadcast its value. After the broadcast, every honest party will share the same vector of values $V = \{v_1, v_2, \cdots, v_n\}$. If party i is honest, then v_i will be its input value. Each party can then just output the most frequent value in the vector V. Agreement is reached since all honest parties share the same V. If all honest parties start with the same input value, that value will be the most frequent in V, achieving validity. As a natural optimization, after the first iteration, we can elect a single leader for all the parallel broadcast instances. Thus, the agreement protocol will have the same round complexity as the broadcast protocol.

6 Conclusion

This paper has described a 4-round synod protocol that tolerates f Byzantine faults using $2f+1$ replicas in the synchronous and authenticated setting. We then apply the synod protocol to solve BFT state machine replication and Byzantine agreement. Our results may be applied to build practical synchronous Byzantine fault tolerant systems and improve cryptographic protocols.

Acknowledgements

The authors thank Dahlia Malkhi for helpful discussions. This work is funded in part by NSF award #1518765 and a Google Ph.D. Fellowship award.

References

[1] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 59–74. ACM, 2005.

[2] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus. arXiv preprint arXiv:1612.02916, 2016.

[2] In fact, we can improve upon Katz and Koo [17]. Their random leader election invokes a 4-round *gradecast*, while our synod protocol without the status round can serve as a 3-round gradecast.

[3] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

[4] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM, 1983.

[5] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 1–10. ACM, 1988.

[6] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[7] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 173–186. USENIX Association, 1999.

[8] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[9] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190. USENIX Association, 2006.

[10] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[11] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[12] Pesech Feldman and Silvio Micali. An optimal probabilistic protocol for synchronous byzantine agreement. *SIAM Journal on Computing*, 26(4):873–933, 1997.

[13] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information processing letters*, 14(4):183–186, 1982.

[14] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[15] Juan A Garay and Yoram Moses. Fully polynomial byzantine agreement for $n > 3f$ processors in $f + 1$ rounds. *SIAM Journal on Computing*, 27(1):247–290, 1998.

[16] Shafi Goldwasser, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with an honest majority. In *Proc. of the 19th Annual ACM STOC*, volume 87, pages 218–229, 1987.

[17] Jonathan Katz and Chiu-Yuen Koo. On expected constant-round protocols for byzantine agreement. *J. Comput. Syst. Sci.*, 75(2):91–112, 2009.

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

[19] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[20] Ramakrishna Kotla and Mike Dahlin. High throughput byzantine fault tolerance. In *Dependable Systems and Networks, International Conference on*, pages 575–584. IEEE, 2004.

[21] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[22] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[23] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[24] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[25] Yehuda Lindell, Anna Lysyanskaya, and Tal Rabin. Sequential composition of protocols without simultaneous termination. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 203–212. ACM, 2002.

[26] Shengyun Liu, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: practical fault tolerance beyond crashes. *CoRR, abs/1502.05831*, 2015.

[27] J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

[28] Silvio Micali. Algorand: The efficient and democratic ledger. arXiv preprint arXiv:1607.01341, 2016.

[29] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17. ACM, 1988.

[30] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. Cryptology ePrint Archive, Report 2016/917, 2016.

[31] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[32] Michael O. Rabin. Randomized byzantine generals. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, pages 403–409. IEEE, 1983.

[33] Michael K. Reiter, Matthew K. Franklin, John B. Lacy, and Rebecca N. Wright. The Ω key management service. In *Proceedings of the 3rd ACM conference on Computer and communications security*, pages 38–47. ACM, 1996.

[34] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. Base: Using abstraction to improve fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 15–28. ACM, 2001.

[35] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[36] Dan S Wallach, Peter Druschel, et al. Enforcing fair sharing of peer-to-peer resources. In *International Workshop on Peer-to-Peer Systems*, pages 149–159. Springer, 2003.

[37] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. *ACM SIGOPS Operating Systems Review*, 37(5):253–267, 2003.

[38] Lidong Zhou, Fred B Schneider, and Robbert Van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, 2002.