Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR

Oded Naor
Technion – Israel Institute of Technology, Haifa, Israel

Idit Keidar
Technion – Israel Institute of Technology, Haifa, Israel

Abstract

State Machine Replication (SMR) solutions often divide time into rounds, with a designated leader driving decisions in each round. Progress is guaranteed once all correct processes *synchronize* to the same round, and the leader of that round is correct. Recently suggested Byzantine SMR solutions such as HotStuff, Tendermint, and LibraBFT achieve progress with a linear message complexity and a constant time complexity once such round synchronization occurs. But round synchronization itself incurs an additional cost. By Dolev and Reischuk's lower bound, any deterministic solution must have $\Omega(n^2)$ communication complexity. Yet the question of randomized round synchronization with an expected linear message complexity remained open.

We present an algorithm that, for the first time, achieves round synchronization with expected linear message complexity and expected constant latency. Existing protocols can use our round synchronization algorithm to solve Byzantine SMR with the same asymptotic performance.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases Distributed Systems, State Machine Replication

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.26

Related Version A full version of the paper is available at [33], https://arxiv.org/abs/2002.07539.

1 Introduction

Byzantine *State Machine Replication (SMR)* has received a lot of attention in recent years due to the increasing demand for robust and scalable systems. In order to tolerate periods of high load or even denial-of-service attacks, practical solutions commonly assume the *eventual synchrony* model [20], meaning that they guarantee consistency despite asynchrony and make progress during periods when the network is synchronous. Examples of such systems include PBFT [16], SBFT [25], LibraBFT [4], HotStuff [37], Zyzzyva [29], Tendermint [13], and many more. Eventually synchronous SMR solutions typically iterate through a sequence of *rounds*, (also called views), wherein a designated *leader* process tries to drive all correct processes to consensus. The main complexity of such algorithms arises whenever a new round begins and its (new) leader collects information about possible consensus decisions in previous rounds.

When using such protocols, it is common to constantly advance in rounds with a rotating leader [16, 37, 4]; this is because when the leader is faulty, it is possible for some processes to perceive progress while others made no progress.

In the last couple of years, there has been a race to improve the performance of Byzantine SMR. Recent algorithms such as Tendermint [13], Casper [14], HotStuff [37], and LibraBFT [4] allow rounds to advance (and leaders to be replaced) with a constant time complexity and a linear message complexity. Thus, even if every consensus instance is led by a different leader, the message complexity for each decision remains linear. Nevertheless, the linear message complexity is achieved only *after* all correct processes synchronize to execute the same round of the protocol (provided that that round's leader is correct). And such *round synchronization* has a cost of its own. In Tendermint, round advancement is gossiped

© Oded Naor and Idit Keidar;
licensed under Creative Commons License CC-BY
34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 26; pp. 26:1–26:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

throughout the system, entailing an expected $O(n \log n)$ message complexity with expected $O(\log n)$ latency. In HotStuff, it is delegated to a separate round synchronization module called *PaceMaker*, whose implementation is left unspecified. And in LibraBFT, this module is implemented with quadratic message complexity, which in fact matches Dolev and Reischuk's $\Omega(n^2)$ communication complexity lower bound [19] on deterministic Byzantine consensus. Later work on Cogsworth [32] implemented a randomized PaceMaker with expected constant latency and linear message complexity under benign failures, but with expected quadratic message complexity in the Byzantine case.

In this work we present a new round synchronization algorithm that achieves expected constant time complexity and expected linear communication complexity even in the presence of Byzantine processes. Specifically, under an oblivious adversary, we guarantee these bounds on the expected time/message cost until all processes synchronize to the same round from an *arbitrary* state of the protocol. Under a strong adversary, we achieve the same bounds but on the *average* expected time and message cost until round synchronization over all states occurring in an infinite run of the protocol. To this end, we decompose the round synchronization module into a *synchronizer* abstraction and two local functions. The synchronizer abstraction captures the essence of the distributed coordination required in order to synchronize processes to the same round.

Like previous works [32, 37, 4], the main technique used in our algorithm to lower the message complexity is a relay-based message distribution with threshold signatures. Instead of broadcasting messages all-to-all, our algorithm sends each message to a designated *relay*. The relay aggregates messages from multiple processes, and when a certain threshold is met, it combines them into a threshold signature, which it sends to all the processes. Note that a threshold signature's size is the same as the original signature sizes, i.e., remains constant as the number of processes grows. This leads to linear communication complexity per message. The challenge is that the relay can be Byzantine and, for example, send the aggregated message only to a subset of the processes. Another challenge arises when some correct process advances to a new round while others lag behind. We introduce a relay-based linear-complexity helping mechanism to allow lagging processes to catch up with faster ones without all-to-all broadcast.

In summary, the main contribution of this paper is providing an algorithm that for the first time reduces the expected message complexity of Byzantine SMR in the presence of Byzantine faults to linear, while maintaining expected constant latency. The rest of the paper is structured as follows: §2 describes the model; §3 formally defines the round synchronization problem and our performance metrics; §4 explains our decomposition of round synchronization into a synchronizer abstraction and local functions, and proves that this decomposition solves the round synchronization problem; §5 presents our new synchronizer algorithm and proves its expected linear message complexity, expected constant latency, and correctness; §6 gives related work and §7 concludes the paper. Some formal proofs are deferred to the full version [33].

2 Model

Our model consists of a set $\Pi = \{\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_n\}$ of n processes. Every two processes in Π have a bidirectional, reliable, and authenticated link between them, i.e., every process can send a message to another process that will eventually arrive and the recipient can verify the sender's identity. We use the term *broadcast* to indicate sending a message to all processes.

We follow the eventually synchronous model [20] in which there is no global clock, and every execution is divided into two periods: first, an unbounded period of asynchrony; and then, a period of synchrony, where messages arrive within a bounded time, δ. The second period begins at a moment called the *Global Stabilization Time (GST)*. Messages sent before GST arrive by GST $+ \delta$. We assume that after GST, processes can correctly estimate such an upper bound δ on communication latency and also measure time locally, but this does not imply a global clock. We consider a failure model where $f < n/3$ processes may be *faulty*, or *Byzantine* and act arbitrarily.

We assume a shared source of randomness, $\mathcal{R}$, that is used to derive a function $\textsc{Relay}(r, k) \colon \mathbb{N} \times \{1, \ldots, f+1\} \mapsto \Pi$. This function is used to select for each round r the k-th process that will act as a relay. The relay function satisfies the following properties:

R1 $f + 1$ different relays for each round:

$$\forall r \in \mathbb{N}, \forall 1 \leq i < j \leq f+1 \colon \textsc{Relay}(r, i) \neq \textsc{Relay}(r, j).$$

R2 Random relay selection, while ensuring $f + 1$ different relays for each round:

$$\forall r \in \mathbb{N}, \forall 1 \leq k \leq f+1, \forall \mathcal{P}_1, \mathcal{P}_2 \in \Pi \setminus \bigcup_{i=1}^{k-1} \{\textsc{Relay}(r, i)\} : \\ \Pr[\textsc{Relay}(r, k) = \mathcal{P}_1] = \Pr[\textsc{Relay}(r, k) = \mathcal{P}_2].$$

Note that R2 implies that the first relay is continuously rotated throughout the run, i.e., $\forall r \in \mathbb{N} \colon \bigcup_{i=r}^{\infty} \textsc{Relay}(i, 1) = \Pi$. Generating secure randomness as assumed by our protocol has been studied in the literature, e.g., [8, 17, 31, 2], and is beyond the scope of this paper.

For clarity of the algorithm's presentation, we assume that the adversary is a *static oblivious adversary* [10, 5, 21], i.e., has no knowledge of the randomness $\mathcal{R}$. This assumption is required for the worst-case performance bounds as defined in §3.2, and can be relaxed to a *strong static adversary* if we only wish to prove an average-case bound. We discuss this in §5.5 below.

Like previous linear-complexity BFT algorithms [4, 13, 32, 37], we use a cryptographic signing scheme, a public key infrastructure (PKI) to validate signatures, and a threshold signing scheme [9, 15, 36]. The threshold signing scheme is used in order to create a compact-sized signature of K-of-N processes as in other consensus protocols [15]. Usually $K = f + 1$ or $K = 2f + 1$. The size of a threshold signature is constant and does depend on K or N. We assume that the adversary is polynomial-time bounded, i.e., the probability that it will break the cryptographic assumptions in this paper (e.g., the cryptographic signatures, threshold signatures, etc.) is negligible.

3 Problem Definition - Round Synchronization

We start by specifying the round synchronization problem in §3.1, then discuss performance metrics in §3.2, and conclude by describing how to use a round synchronization module to solve consensus in §3.3.

3.1 Specification

We define a long-lived task of *round synchronization*, parameterized by the desired round duration Δ. It has a single output signal at process $\mathcal{P}_i$, $\mathsf{round_leader}_i(r, \mathcal{P})$, $r \in \mathbb{N}, \mathcal{P} \in \Pi$, indicating to $\mathcal{P}_i$ to enter round r of which $\mathcal{P}$ is the leader. We say that a process $\mathcal{P}_i$ is *in*

round r between the time t when $\mathsf{round_leader}_i(r, \cdot)$ occurs and the next $\mathsf{round_leader}_i(r', \cdot)$ event after t. If no such event occurs, $\mathcal{P}_i$ remains in round r from t onward. The goal of round synchronization is to reach a *synchronization time*, defined as follows:

▶ **Definition 3.1** (Synchronization time). *Time t_s is a* synchronization time *if all correct processes are in the same round r from t_s to at least $t_s + \Delta$, and r has a correct leader.*

A round synchronization module satisfies two properties. The first ensures that in every round all the correct processes have the same leader.

▶ **Property 1** (Leader agreement). *For any two correct processes $\mathcal{P}_i, \mathcal{P}_{i'}$ if $\mathsf{round_leader}_i(r, \mathcal{P}_j)$ and $\mathsf{round_leader}_{i'}(r, \mathcal{P}_{j'})$ occur, then $\mathcal{P}_j = \mathcal{P}_{j'}$.*

The second property ensures that synchronization times eventually occur. Formally:

▶ **Property 2** (Eventual round synchronization). *For every time t in a run, there exists a synchronization time after t.*

3.2 Performance Metrics

For an oblivious adversary, we measure the maximum expected performance after GST under all possible adversary behaviors and protocol states, where the expectation is taken over random outputs of our randomness source $\mathcal{R}$, which drives the relay function. In more detail, let S be the set of all reachable states of a round synchronization algorithm, and let $\mathcal{A}$ be the set of all possible adversary behaviors after GST. This includes selecting up to f processes to corrupt and scheduling all message deliveries within at most δ time. For a state $s \in S$, and adversary behavior $a \in \mathcal{A}$, let $RS(s, a, \pi)$ be the time from when s occurs until the next synchronization time in a run extending s with adversary behavior a and the relay function derived from the random bits $\pi \in \mathcal{R}$.

The *worst-case expected latency* of the round synchronization module is defined as

$$\max_{\substack{s \in S \\ a \in \mathcal{A}}} \left\{ \mathop{\mathbb{E}}_{\pi \in \mathcal{R}} [RS(s, a, \pi)] \right\}.$$

Similarly, to define message complexity let $M(s, a, r)$ be the total number of messages correct processes send from state s until the next synchronization time in a run extending s with adversary $a \in \mathcal{A}$ and relay output $\pi \in \mathcal{R}$. The *worst-case message complexity* is defined as

$$\max_{\substack{s \in S \\ a \in \mathcal{A}}} \left\{ \mathop{\mathbb{E}}_{\pi \in \mathcal{R}} [M(s, a, \pi)] \right\}.$$

For brevity, in the rest of this paper, we omit the parameters s, a, π, and simply bound the expected latency or message cost over all reachable states and adversary behaviors.

3.3 Using Round Synchronization to Solve Consensus

In HotStuff [37], Theorem 4 states the following in regards to reaching a decision in the consensus protocol:

> "After GST, there exists a bounded time period T_f such that if all correct replicas remain in view v during T_f and the leader for view v is correct, then a decision is reached."

The round synchronization module satisfies exactly the conditions of the theorem, i.e., an eventual round that all the correct processes are in at the same time for at least $\Delta = T_f$, and the leader of that round is correct.

Given a round synchronization module with expected linear message complexity and expected constant latency, HotStuff solves consensus in the same expected asymptotic message complexity and latency as the round synchronization module. In addition, HotStuff also uses the same cryptographic primitives (namely threshold signatures) as we use in this paper, incurring similar computational costs.

Note that, in general, processes know neither whether their leader is correct nor whether all correct processes are in the same round as them. Indeed, it is possible for a set of $f+1$ correct processes (and f Byzantine ones) to make progress in a round with a Byzantine leader, while f correct processes are stuck behind. In an SMR algorithm where the processes communicate only with the leader of each round and do not broadcast decisions to all processes, this scenario is indistinguishable from one where the leader is correct and all correct processes make progress. Therefore, to ensure the condition required by HotStuff (and captured by Property 2), we continuously advance in rounds and change leaders, regardless of the observed progress made in the consensus protocol utilizing the round synchronization module.

4 Round Synchronization Decomposition

We build the round synchronization module using a *synchronizer* abstraction and two local modules. The synchronizer captures the necessary distributed coordination among the processes. The abstraction's properties appear in §4.1, and a round synchronization module using this abstraction is given in §4.2. The latter consists of a *timer* function that paces the synchronizer and a *leader* function that outputs the leader and round to the application. This decomposition is illustrated in Figure 1.

4.1 Synchronizer

We define a *synchronizer* abstraction to be a long-lived task with an API that includes an $\mathit{advance}_i()$ input and a $\mathit{new_round}_i(r)$ output signal, where $r \in \mathbb{N}$.

In a similar way to the round synchronization module, we say that process $\mathcal{P}_i$ *enters round* r when $\mathsf{new_round}_i(r)$ occurs. We say *process* $\mathcal{P}$ *is in round* r during the time interval that starts when $\mathcal{P}$ enters round r and ends when it next enters another round. If the process does not enter a new round, then it remains indefinitely in r. We denote by $r_max(t)$ the maximum round a correct process is in at time t.

We define four properties a synchronizer algorithm should guarantee. The first ensures that rounds are monotonically increasing. Formally:

▶ **Property 3** (Monotonically increasing rounds)**.** *For each correct process* $\mathcal{P}_i$*, if* $\mathit{new_round}_i(r')$ *occurs after* $\mathit{new_round}_i(r)$*, then* $r' > r$*.*

The next property is the validity of new rounds.

▶ **Property 4** (Validity)**.** *If a correct process signals* $\mathit{new_round}(r)$ *then some correct process called* $\mathit{advance}()$ *while in round* $r-1$*.*

Next, we define the two liveness properties. Informally, the first ensures the stabilization of at least $f+1$ correct processes to the same maximum round, and the second ensures progress after the stabilization.

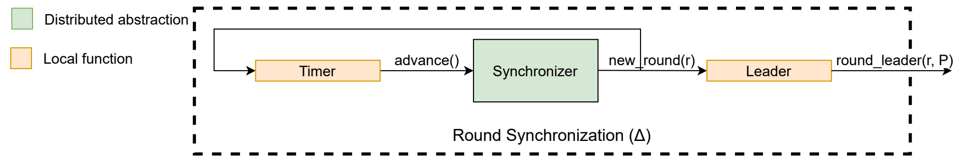

Figure 1 Round synchronization using the synchronizer abstraction.

Algorithm 1 Round synchronization using the synchronizer abstraction.

```
1 Timer:
2     after c1 + Δ from last new_round(r) signal: // c1 is defined in S2
3         advance()
4 Leader:
5     on new_round(r) signal:
6         round_leader(r, Relay(r, 1))
```

▶ **Property 5** (Stabilization). *For any t during the run, let t_0 be the first time when a correct process enters round $r_max(t)$. If no correct process enters any round $r > r_max(t)$, then:*

S1 *From some time t_1 onward, at least $f + 1$ correct processes are in round $r_max(t)$.*

S2 *If $t_0 \geq GST$ and $\text{Relay}(r_max(t), 1)$ is correct, then from some time t_2 onward all the correct processes enter $r_max(t)$ and $t_2 - t_0 \leq c_1$ for some constant c_1.*

Although Property 5 is primed on no correct processes *ever* entering rounds higher than $r_max(t)$, we observe that S2 holds as long as no process enters rounds higher than $r_max(t)$ by $t_0 + c_1$ because any such run is indistinguishable to all processes until time $t_0 + c_1$ from a run where they never enter a higher round at all. Formally:

▶ **Observation 1.** *Assume Property 5 holds, then for any t during the run, let $t_0 \geq GST$ be the first time when a correct process enters round $r_max(t)$. If no correct process enters any round $r > r_max(t)$ by $t_0 + c_1$ for some constant c_1 and $\text{Relay}(r_max(t), 1)$ is correct, then all correct processes enter round $r_max(t)$ by $t_0 + c_1$.*

The next property ensures progress.

▶ **Property 6** (Progress). *For any t during the run, if $f + 1$ correct processes in round $r_max(t)$ call advance() by t_0, and no correct process calls advance() while in any round $r > r_max(t)$ then:*

P1 *From some time t_1 onward, there is at least one correct process in $r_max(t) + 1$.*

P2 *If $t_0 \geq GST$ and $\text{Relay}(r_max(t), 1)$ is correct, then from some time t_2 onward all the correct processes enter $r_max(t) + 1$ and $t_2 - t_0 \leq c_2$ for some constant c_2.*

Property P2 is not required for round synchronization, but it gives a bound on performance.

4.2 From Synchronizer to Round Synchronization

We now describe how to use the synchronizer abstraction to implement round synchronization. The implementation uses two local functions: a *timer* function that paces a process' advance() calls, and a *leader* function that maps a round to a leader using the Relay function. This construction is illustrated in Figure 1, and specified in Alg. 1. When one module invokes a function in another, we refer to this as a *signal*, e.g., the timer signals advance() to the synchronizer.

We prove that this construction provides round synchronization. Let t^0 = GST and $\forall \ell \geq 1$ let t^ℓ be the first time after $t^{\ell-1}$ that a correct process enters a new maximum round. We prove the following lemma:

▶ **Lemma 4.1.** *In an infinite run of Alg. 1, t^ℓ eventually occurs for any $\ell \geq 0$.*

Proof. We prove this by induction on ℓ. Based on the model, the base step of the induction, t^0 = GST eventually occurs.

Next, assume that t^ℓ occurs during the run. If $t^{\ell+1}$ occurs, then we are done.

Assume by contradiction that $t^{\ell+1}$ does not occur, i.e., by the induction hypothesis some correct process entered $r_max(t^\ell)$ but no correct process enters any round $r > r_max(t^\ell)$. By S1, eventually at least $f + 1$ correct processes enter $r_max(t^\ell)$. Denote this set of processes by P. The timer function ensures that eventually every process in P calls advance(), so there are at least $f + 1$ correct processes in $r_max(t^\ell)$ that call advance(). By P1, eventually at least one correct process enters $r_max(t^{\ell+1}) = r_max(t^\ell) + 1$, a contradiction to the assumption that no correct process enters any round $r > r_max(t^\ell)$. ◀

We prove the main theorem of this section:

▶ **Theorem 4.2.** *Using a synchronizer abstraction, Alg. 1 implements a round synchronization module.*

Proof. Since the relay function's outputs are identical among all correct processes and the leader local function outputs round_leader(r, Relay(r, 1)), it is immediate that the leader agreement property (Property 1) is satisfied.

We now prove eventual round synchronization (Property 2). Define $Leader(\ell) \triangleq$ Relay($r_max(t^\ell)$, 1). By Lemma 4.1, t^i occurs for all $i \geq 0$, and since the first relay for each round is randomly chosen, eventually, with probability 1, there exists a $\ell \geq 0$ such that $Leader(\ell)$ is a correct process. Let us look at $r_max(t^\ell)$, and denote $\widetilde{t} \triangleq t^\ell + c_1 + \Delta$.

Recall that t^ℓ is the time when the first correct process enters $r_max(t^\ell)$. By Line 2 in Alg. 1, no correct process calls advance() between t^ℓ and $\widetilde{t}$, and because of validity (Property 4) no correct process enters any round $r > r_max(t^\ell)$ until at least $\widetilde{t}$. By using Observation 1, we can apply S2 for $r_max(t^\ell)$, since by $t^\ell + c_1$ all correct processes enter $r_max(t^\ell)$.

Thus, between t^ℓ and $t^\ell + c_1$, all correct processes enter $r_max(t^\ell)$. Since no correct process calls advance() until at least $\widetilde{t}$, this guarantees that all correct processes remain in $r_max(t^\ell)$ until $\widetilde{t} = t^\ell + c_1 + \Delta$, so $t^\ell + c_1$ is a synchronization time (Def. 3.1), as needed. ◀

5 An Expected Linear Message Complexity and Constant Latency Synchronizer

In this section we present a synchronizer abstraction algorithm with expected linear message complexity and constant latency in the Byzantine case.

We start by describing the main ideas used to lower the message complexity (while still guaranteeing constant latency) in §5.1. We give a more in-depth description of the algorithm in §5.2, reason about the algorithm's correctness in §5.3, and performance in §5.4. Some of the formal proofs are deferred to the full version [33].

5.1 Achieving Linear Message Complexity

The crux of the algorithm is a relay-based distribution of messages among processes. A standard Byzantine broadcast system, which ensures that a message sent by a correct process is eventually delivered by all other correct processes, usually requires quadratic message

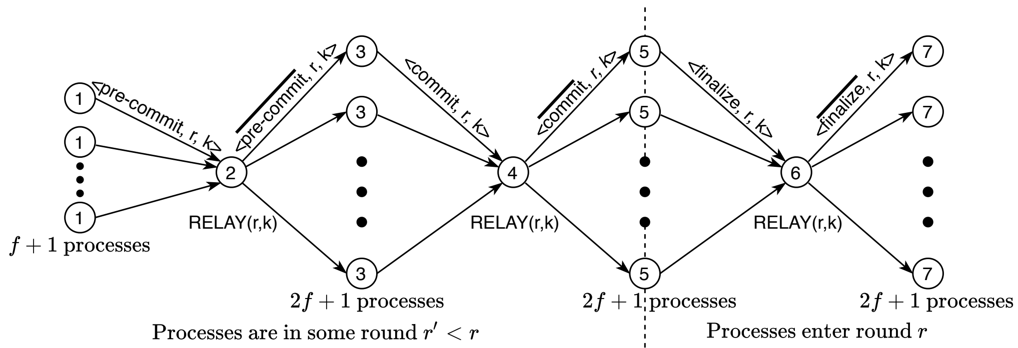

Figure 2 The message flow of the algorithm. A process enters round r when it receives a $\overline{\text{commit}}$ message for round r, the circled numbers represent the different stages of the algorithm (see Alg. 2). Only the $2f + 1$ correct processes are illustrated.

complexity for each message disseminated. This is because in Byzantine broadcast protocols such as Bracha's [11], when a correct process delivers a message, it also sends it to all the other processes, resulting in all-to-all communication for each delivered message.

In our algorithm, we instead use a single designated process as a relay. Processes send their messages to the relay, which aggregates messages from a number of processes, combines them into one message using a threshold signature, and broadcasts it to all the processes. This mechanism reduces the total number of protocol messages from $O(n^2)$ to $O(n)$.

A difficulty arises if the relay is Byzantine. We overcome this as follows: when a process $\mathcal{P}$ sends a message to a relay, it expects a response from it within a certain time-bound. If no timely response arrives, $\mathcal{P}$ can deduce that either GST has not occurred yet and the message to/from the relay is delayed, or it is after GST and the relay is Byzantine. In either case, after the allotted time passes, $\mathcal{P}$ proceeds to send a message to a different relay, again waiting for the new relay to respond in a timely manner, and so on. This mechanism uses the relay function described in §2. Once a correct relay is contacted, the algorithm makes progress. In expectation, the number of consecutive Byzantine relays until a correct one is bounded by $3/2$, leading to expected constant latency and linear message complexity. In the worst-case, each round has $f + 1$ potential relays, guaranteeing that at least one of them is correct, which ensures liveness.

5.2 Algorithm Description

At a high level, the goal of the algorithm is to eventually enter all rounds during the run, and reach a synchronization time after GST in every round r where $\textsc{Relay}(r, 1)$ is a correct process. If the relay is Byzantine, then the goal is to eventually move from r to $r + 1$. The randomization of the relay function guarantees that in an infinite run there will be infinitely many rounds with a correct process as the first relay, guaranteeing an infinite number of synchronization times.

Message flow of the algorithm. The algorithm is presented in Alg. 2, and its message flow is depicted in Figure 2. Protocol messages are signed and verified; for brevity, we omit the signatures and their verification from the algorithm description and pseudocode.

A process sends to $\textsc{Relay}(r, k)$ messages of the form $\langle \text{message type}, r, k \rangle$, where r and k are natural numbers, and message type is one of the following: pre-commit, commit, or finalize. The relay's messages to the processes are threshold signatures on an aggregation of the same messages, denoted $\langle \overline{\text{pre-commit}}, r, k \rangle$, $\langle \overline{\text{commit}}, r, k \rangle$, and $\langle \overline{\text{finalize}}, r, k \rangle$, respectively. Each threshold signature is created using some number ($f+1$ or $2f+1$) of signatures.

When advance() is signaled via the local timer function (see §4.2) to indicate that it wants to move from round $r-1$ to round r, the process sends a pre-commit message to the relay (this is stage 1 of the algorithm). Once $f+1$ processes indicate that they wish to move to round r, the relay broadcasts a $\overline{\text{pre-commit}}$ message (stage 2). The reason $f+1$ processes are needed to initiate the first stage of the algorithm is to ensure that there is at least one correct process among them, preventing Byzantine processes from causing correct ones to advance prematurely. Any process receiving a relay's $\overline{\text{pre-commit}}$ message in a round $r' < r$ joins in by sending a commit message for r (stage 3). Unlike in previous work such as Cogsworth [32], $\overline{\text{pre-commit}}$ messages are linked to a particular relay, and therefore, if the protocol times out and proceeds to the next relay, the new relay needs to collect $f+1$ pre-commits afresh. This subtle difference prevents Byzantine relays from spuriously engaging in the protocol, which is crucial for avoiding the quadratic message complexity occurring in Cogsworth.

When $2f+1$ processes indicate that they commit to moving to r, the relay sends a $\overline{\text{commit}}$ message (stage 4) and processes that receive it enter that round (stage 5). Requiring $2f+1$ processes to commit to a round r before entering it ensures that at least $f+1$ correct processes are aware of the intent to enter r. This ensures that at least $f+1$ correct processes will eventually enter r, and those $f+1$ processes guarantee progress, as it is the minimal quorum required to initiate the stages of the algorithm to the next round, until a round with a first correct relay is reached and in that round a synchronization time will occur.

However, the algorithm for synchronizing for round r does not end when a process receives a $\overline{\text{commit}}$ message for r. Rather, a process that enters round r sends a finalize message to help any lagging processes with the transition to round r. Once $2f+1$ finalize messages are sent, the relay broadcasts a $\overline{\text{finalize}}$ message (stage 6), and when a process receives it, it completes the algorithm for round r (stage 7). The finalization phase is needed to overcome cases of a Byzantine relay that does not send the $\overline{\text{commit}}$ message to all the processes.

Variables and timeouts. The variable *curr_round* stores the current round a process is currently in which changes in stage 5, and *next_round* indicates to what round the process is attempting to enter. The value of *next_round* becomes *curr_round* + 1 when a process invokes advance(), and it can become higher if the process learns (via a $\overline{\text{pre-commit}}$) of at least $f+1$ other processes that want to advance to a higher round than the one the process is currently in.

The timeouts at the bottom of the pseudocode dictate when a process moves to the next relay of a round. When a process sends a message to a relay, it expects the relay to respond within 2δ, which is the upper bound of the round-trip time after GST. For example, if a process sends a message of round r to $\textsc{Relay}(r, k)$ at time t and does not receive a response by $t + 2\delta$, it sends the message to $\textsc{Relay}(r, k+1)$. This continues up to $\textsc{Relay}(r, f+1)$, guaranteeing that at least one of the relays for round r is correct.

Upon a timeout, a process sends a pre-commit message to the next relay in line, and once that relay gets f+1 such messages, it, too, can try to complete the stages of the protocol for the same round. There is a tradeoff involved in choosing the timeout – a shorter timeout may cause a second relay to engage even when the first relay is correct, whereas a longer one delays progress in case of a Byzantine relay. Nevertheless, it is important to note that a

process responds to all relays, so contacting the $(k+1)$-st relay for round r does not in any way prevent the k-th one from making progress. Thus, while setting an aggressive timeout may cause the protocol to send more messages, it does not in any way hamper progress. A process that partakes in the protocol to advance to round r contacts a new relay every 2δ time for as long as it does not make progress in the phases of the algorithm for round r. Since a process takes an expected 6δ to complete the algorithm for round r, the process contacts 3 relays in expectation.

The *round_relay* array holds the highest relay for each round the process sent a pre-commit message to. For example, $round_relay[r] = k$ for $k > 1$ indicates that the process sent $\langle \textsf{pre-commit}, r, 1 \rangle, \ldots, \langle \textsf{pre-commit}, r, k \rangle$ messages to $\textsc{Relay}(r, 1), \ldots, \textsc{Relay}(r, k)$, respectively. Note that a process sends a pre-commit message for round r to $\textsc{Relay}(r, 1)$ when it first receives a $\overline{\text{pre-commit}}$ message in stage 3, regardless of the relay it received the message from. This is to allow the first relay of round r to complete the stages of the algorithm in case it is correct, and make sure that round synchronization will occur in round r. Note that the fact that some relay sends a message with a threshold signature does not ensure that that relay is correct, even if all the signatures used to create the threshold signature are from correct processes. For example, a Byzantine relay can broadcast a message only to a subset of the correct processes. Thus, to ensure liveness, processes must iterate through all $f+1$ relays of a round, starting from the first one, until progress is made.

We note that the *round_relay* array is introduced in the pseudocode for simplicity, but in a real implementation there is no need for an unbounded array to be stored in memory. A process only sends messages to the relays of rounds stored in the *curr_round* and *next_round* variables, thus limiting the amount of memory needed for an actual implementation to a constant number of integers.

Example. To clarify the need for the last phase of the algorithm (stages 6 and 7), consider the following scenario: Suppose a set P of $f+1$ correct processes are in round $r-1$ and invoke $\textsf{advance}()$. The remaining f correct processes are in a round $r' < r-1$. The processes in P send a pre-commit message to $\textsc{Relay}(r, 1)$, which is Byzantine. The relay generates a threshold signature and sends a $\overline{\text{pre-commit}}$ only to the processes in P, which respond with a commit message. Now, $\textsc{Relay}(r, 1)$, with the help of f Byzantine processes, creates a $\overline{\text{commit}}$ message for r, but sends it to only one correct process $\mathcal{P}_i$ in P. This results in a scenario where $\mathcal{P}_i$ is the only correct process in round r, while f correct processes remain in round $r-1$ and continue to timeout and send pre-commit messages to the relays of round r. Since a relay needs at least $f+1$ pre-commit messages to engage the stages of the algorithm, unless $\mathcal{P}_i$ continues to help the rest of the processes in P by sending pre-commit messages, they might get stuck in round $r-1$. Therefore, processes continue to timeout and send pre-commit messages in the previous round until they receive a $\overline{\text{finalize}}$ message. Once a process in r receives a $\overline{\text{finalize}}$ message for r, it knows that there are at least $f+1$ correct processes in round r, and can stop sending pre-commit messages for r. This is crucial for achieving the desired message complexity after GST. These $f+1$ correct processes will eventually call $\textsf{advance}()$ and proceed to round $r+1$.

Algorithm 2 Synchronizer Algorithm. The circles show the protocol's stages.

```
1  initialize:
2    curr_round ← 0 // Processes begin their execution at round 0.
3    next_round ← 0
4    ∀i ∈ ℕ: round_relay[i] ← 1
5    finalized ← True

Every process:
(1)
6  on advance() signal:
7    if curr_round < next_round then // old round
8      return
9    next_round ← curr_round + 1
10   send ⟨pre-commit, next_round, 1⟩ to RELAY(next_round, 1)
(3)
13 upon receiving the first valid ⟨pre-commit, r, k⟩ from RELAY(r, k):
14   if r < next_round then // old round
15     return
16   if r > next_round then /* start participating in round r */
17     next_round ← r
18     send ⟨pre-commit, r, 1⟩ to RELAY(r, 1)
19   send ⟨commit, r, k⟩ to RELAY(r, k)
(5)
22 upon receiving the first valid ⟨commit, r, k⟩ from RELAY(r, k):
23   if r < curr_round then // old round
24     return
25   if r > curr_round then // enter round r
26     curr_round ← r
27     finalized ← False
28     send ⟨commit, r, 1⟩ to RELAY(r, 1)
29     new_round(r) // signal new round
30   send ⟨finalize, r, k⟩ to RELAY(r, k)
(7)
33 upon receiving the first valid ⟨finalize, r, k⟩ from RELAY(r, k):
34   if r = curr_round then
35     finalized ← True

Relay (Relay(r, k)):
(2)
11 upon receiving the first valid f + 1 ⟨pre-commit, r, k⟩ messages:
12   broadcast ⟨pre-commit, r, k⟩
(4)
20 upon receiving the first valid 2f + 1 ⟨commit, r, k⟩ messages:
21   broadcast ⟨commit, r, k⟩
(6)
31 upon receiving the first valid 2f + 1 ⟨finalize, r, k⟩ messages:
32   broadcast ⟨finalize, r, k⟩

Timeouts (for every process):
36 on pre-commit and commit timeouts: /* Every 2δ from last sending pre-commit or commit
   messages and not receiving the matching pre-commit or commit */
37   if round_relay[next_round] < f + 1 then
38     round_relay[next_round] ← round_relay[next_round] + 1
39     send ⟨pre-commit, next_round, round_relay[next_round]⟩ to
       RELAY(next_round, round_relay[next_round])
40 on finalize timeout: // Every 2δ from last sending finalize and not receiving the matching finalize
41   if finalized = False and round_relay[curr_round] < f + 1 then
42     round_relay[curr_round] ← round_relay[curr_round] + 1
43     send ⟨pre-commit, curr_round, round_relay[curr_round]⟩ to
       RELAY(curr_round, round_relay[curr_round])
```

5.3 Correctness

Next, we prove that the algorithm satisfies the properties of a synchronizer, as defined in §4.1. The proofs of Lemma 5.1 and Lemma 5.5 are in the full version of the paper.

▶ **Lemma 5.1.** *Alg. 2 satisfies monotonically increasing rounds (Property 3).*

▶ **Lemma 5.2.** *Alg. 2 satisfies round validity (Property 4).*

Proof. A correct process enters round r when it is in a round $r' < r$ and receives a $\overline{\text{commit}}$ message for r. A $\overline{\text{commit}}$ message is a threshold signature of $(2f+1)$-of-n commit messages, meaning at least $f+1$ are from correct processes. A correct process sends a commit message for round r when it receives a $\overline{\text{pre-commit}}$ message for r. A $\overline{\text{pre-commit}}$ message is a threshold signature of $(f+1)$-of-n pre-commit messages, meaning at least one correct process sent a pre-commit message for round r.

Denote $\mathcal{P}_i$ as the first correct process that sends a pre-commit message for r during the run. A correct process only sends a pre-commit for r (in Lines 10, 18, 39, and 43) when its *next_round* or *curr_round* variables hold r. *next_round* changes in one of two places – Line 9 when a process calls advance(), and Line 17 on receiving a valid $\overline{\text{pre-commit}}$ for r. *curr_round* changes on receiving a valid $\overline{\text{commit}}$ for r. Because no $\overline{\text{pre-commit}}$ or $\overline{\text{commit}}$ message can be sent for round r before at least one correct process sends a pre-commit for r, then $\mathcal{P}_i$ must have sent its pre-commit message for round r when it changed its *next_round* in Line 9, i.e., on executing advance(). ◀

▶ **Proposition 5.3.** *If a correct process receives a $\overline{\text{finalize}}$ for round r at time t, then at least $f+1$ correct processes entered round r by t.*

Proof. Let t be a time in which a correct process received a $\overline{\text{finalize}}$ message for round r. This message is a threshold signature of $(2f+1)$-of-n finalize messages, of which at least $f+1$ originated from correct processes. A correct process only sends a finalize message for r if it receives a $\overline{\text{commit}}$ message for r, which means that it is already in round r by time t. ◀

▶ **Lemma 5.4.** *Alg. 2 satisfies stabilization (Property 5) with $c_1 = 4\delta$.*

Proof. Let t be a point in time during the execution and $r = r_max(t)$. Let $\mathcal{P}_i$ be the first correct process that enters round r at time t_0. Such a process exists by the definition of $r_max(t)$. $\mathcal{P}_i$ is at round r, so it received a $\overline{\text{commit}}$ message for round r. A $\overline{\text{commit}}$ message is a threshold signature of $(2f+1)$-of-n commit messages, at least $f+1$ of which were sent by correct processes. Denote by S the set of correct processes whose signatures on commit messages are included in the $\overline{\text{commit}}$ message $\mathcal{P}_i$ received. The processes in S are either in round r at time t or in smaller rounds $r' < r$.

We now prove the two sub-properties of Property 5:

S1. If some correct process receives $\overline{\text{finalize}}$ for round r, by Proposition 5.3, there are at least $f+1$ correct processes in r and we are done.

Assume no correct process receives $\overline{\text{finalize}}$. Then, the processes in S continue to timeout and send pre-commit messages for round r to the relays of r. This guarantees that eventually, a correct relay for r receives at least $f+1$ pre-commit messages, as Property R1 of the relay function ensures $f+1$ different relays for each round. This relay eventually completes the stages of the algorithm, allowing all correct processes to advance to round r.

S2. Because $\mathcal{P}_i$ receives $\overline{\text{commit}}$ for round r at time $t_0 \geq \text{GST}$, as argued above, $f+1$ correct processes have sent a commit message for round r by time t_0. Because a process sends pre-commit to $\textsc{Relay}(r,1)$ before sending a commit to any relay for round r (Lines

10 or 18), these messages, too, are sent by time t_0. Therefore, by time $t_0 + \delta$, $\textsc{Relay}(r, 1)$ receives $f + 1$ pre-commit messages and sends a $\overline{\text{pre-commit}}$ message to all processes. By $t_0 + 2\delta$ all the correct processes receive the $\overline{\text{pre-commit}}$ message sent from the first relay, by $t_0 + 3\delta$ the relay receives $2f + 1$ commit messages (along with any process that already entered r, Line 28), and by $t_2 \leq t_0 + 4\delta$ all the correct processes receive the $\overline{\text{commit}}$ message and enter round r. ◀

▶ **Lemma 5.5.** *Alg. 2 satisfies progress (Property 6) with $c_2 = 4\delta$.*

The following theorem follows directly from Lemmas 5.1, 5.2, 5.4, and 5.5.

▶ **Theorem 5.6.** *Alg. 2 satisfies the synchronizer abstraction.*

5.4 Performance: Latency and Message Complexity

We prove that our algorithm has expected constant latency and linear message complexity. The proofs of Proposition 5.7 and Lemmas 5.8 and 5.9 appear in the full version of the paper.

▶ **Proposition 5.7.** *For any round r, let X_r be the number of consecutive Byzantine relays until the first correct relay. Then, $\forall r \colon \mathbb{E}[X_r] \leq 3/2$.*

▶ **Lemma 5.8.** *For any $t \geq GST$ let t_0 be the first time during the run where a correct process enters $r_max(t)$. There exists a time $t_1 \geq t_0$ such that up to t_1 either (i) at least $f+1$ correct processes are in $r_max(t)$* **or** *(ii) a correct process enters a round $r > r_max(t)$; and $\mathbb{E}[t_1 - \max\{t_0, GST\}] \leq \frac{3}{2} \cdot 6\delta$.*

▶ **Lemma 5.9.** *For any $t \geq GST$ let t_0 be the first time when $f + 1$ correct processes call* $\mathsf{advance}()$ *while in round $r_max(t)$. There exists a time $t_1 \geq t_0$ such that there is at least one correct process in $r_max(t) + 1$ and $\mathbb{E}[t_1 - \max\{t_0, GST\}] \leq \frac{3}{2} \cdot 6\delta$.*

▶ **Theorem 5.10.** *The synchronizer algorithm along with a Timer local function (as defined in §4) achieves expected constant latency and linear message complexity.*

Proof. The latency for our algorithm is based on the definition in §3.2. We go over all possible states after GST the correct processes in our algorithm can be in, and look at the expected latency until the synchronization time. Let $t^0 = \text{GST}$ and for all $\ell \geq 1$ let t^ℓ represent the first time after $t^{\ell-1}$ that a correct process enters a new maximum round. By Lemma 4.1, in an infinite run, t^ℓ eventually occurs for any $\ell \geq 0$. For any time $t \geq \text{GST}$ during the run, let $sync_time(t)$ be the first time after t until a synchronization time (Def. 3.1). To calculate the expected latency of our algorithm, we need to show that for any $t \geq \text{GST}$, $E_1 \triangleq \mathbb{E}[sync_time(t) - t] \leq O(\delta)$.

Denote E_2 as the expected time from any time $t \geq \text{GST}$ until the next t^ℓ, i.e., for any $l \geq 0$ and t, $E_2 \triangleq \mathbb{E}\left[\min_{t^\ell \geq t}\left\{t^\ell\right\} - t\right]$ and $E_3 \triangleq \mathbb{E}\left[t^{\ell+1} - t^\ell\right]$. If $\textsc{Relay}(r_max(t^\ell), 1)$ is correct, then based on P2, by $t^\ell + 4\delta$ all the correct processes enter $r_max(t^\ell)$. Therefore:

$$E_1 \leq E_2 + \underbrace{\frac{n-f}{n}}_{\substack{\text{Probability that}\\ \textsc{Relay}(r_max(t^\ell),1)\\ \text{is correct}}} \cdot \underbrace{4\delta}_{\substack{\text{The maximum time for}\\ \text{all correct processes}\\ \text{to enter a round}\\ \text{(Lemma 5.4)}}} + \underbrace{\frac{f}{n}}_{\substack{\text{Probability that}\\ \textsc{Relay}(r_max(t^\ell),1)\\ \text{is Byzantine}}} \cdot \underbrace{\mathbb{E}\left[sync_time(t^{\ell+1}) - t^\ell\right]}_{\substack{\text{The expected time until all}\\ \text{correct processes}\\ \text{enter } r_max(sync_time(t^{\ell+1}))}} =$$

$$= E_2 + \frac{n-f}{n} \cdot 4\delta + \frac{f}{n} \cdot \Big(E_1 + \underbrace{\mathbb{E}\left[t^{\ell+1} - t^\ell\right]}_{=E_3}\Big)$$

$$\Rightarrow E_1 \leq \frac{n}{n-f}\left(E_2 + \frac{n-f}{n} \cdot 4\delta + \frac{f}{n} \cdot E_3\right). \tag{1}$$

Assuming that once a correct process enters a new round, the timer calls advance() within $4\delta + \Delta$, the expected time between t^ℓ and $t^{\ell+1}$ can be bounded as follows:

$$E_3 = \mathbb{E}\left[t^{\ell+1} - t^\ell\right] \leq \underbrace{\mathbb{E}\left[\begin{array}{c}\text{Time from } t^\ell \text{ until} \\ \text{at least } f+1 \text{ correct} \\ \text{processes enter } r_max(t^\ell) \\ \text{or } t^{\ell+1} \text{ occurs}\end{array}\right]}_{\text{Lemma 5.8}} + \underbrace{4\delta + \Delta}_{\substack{\text{Time until at least } f+1 \\ \text{correct processes call} \\ \text{advance() in } r_max(t^\ell)}} + \underbrace{\mathbb{E}\left[\begin{array}{c}\text{Time from the first} \\ \text{time } f+1 \text{ correct} \\ \text{processes call advance()} \\ \text{in } r_max(t^\ell) \text{ and} \\ \text{until } t^{\ell+1} \text{ occurs}\end{array}\right]}_{\text{Lemma 5.9}}$$
$$\leq \frac{3}{2} \cdot 6\delta + 4\delta + \Delta + \frac{3}{2} \cdot 6\delta = 22\delta + \Delta.$$

The calculation of E_3 proves that in expectation, the time between any t^ℓ and $t^{\ell+1}$ is expected constant, assuming Δ is constant. Therefore, E_2 is also expected constant.

To conclude, we proved that $E_2 \leq O(\delta)$ and $E_3 \leq O(\delta)$, and by Eq. (1), $E_1 \leq O(\delta)$, as needed to prove expected constant latency.

For the message complexity of the synchronizer, note that since the expected time between two occurrences of round synchronization is expected constant, the message complexity is expected linear. This is because for a given round the number of consecutive Byzantine relays until a correct one is expected constant, and in the algorithm, every process sends one message to the relay in each stage of the algorithm, and the relay responds with one message to all the processes. Even if a process contacts more than one relay per round, it still contacts an expected constant number of relays, and therefore this does not hamper the asymptotic linear message complexity. ◀

5.5 Relaxed Model

As part of the model in §2 we assumed that the adversary is oblivious. If the adversary is *strong*, and knows the randomness $\mathcal{R}$ before choosing which processes to corrupt, a worst-case bound is tantamount to a deterministic one, because it holds for all coin flips. Therefore, we cannot hope to get a linear message complexity for the worst-case. Nevertheless, in a run with infinitely many round synchronization events, we can bound the *average-case* expected latency and message complexity by considering the limit of the average latency and message complexity on prefixes of length t of the run as t tends to infinity.

Thus, a strong adversary who is aware of the relay function, can choose to corrupt, e.g., the first f relays of some round r, causing that round to have linear latency and quadratic message complexity. But since the adversary is static, it has to corrupt the same processes in all rounds, and by property R2, this does not impact the average-case performance.

6 Related Work

Algorithms for the eventual synchrony model almost invariably use the notion of round or views [30, 34, 26, 6]. A number of works have suggested frameworks and mechanisms for round synchronization in the benign case [3, 23, 27, 28, 24]. For example, Awerbuch introduced synchronizers [3] for failure-free networks. TLC [23] places a barrier on round advancement, so that processes enter round $r+1$ only after a threshold of the processes entered round r. Frameworks like RRFD [24] and GIRAF [27, 28] create a round-based structure for eventually synchronous and failure-detector based algorithms.

A related concurrent work due to Bravo et al. [12] also tackles the liveness of consensus protocols, and creates a general framework to abstract the liveness part of consensus protocols. They show that some protocols such as PBFT [16] and HotStuff [37] can use this framework.

They also provide an algorithm for round synchronization that, starting from some round r, synchronizes all rounds $r' \geq r$ (even rounds with a Byzantine leader, in which decisions are not made), but unlike our algorithm, it requires a quadratic communication cost per synchronization event. They assume a relaxed network model compared to us, where before GST messages might be lost.

Several algorithms include two modes of operation: a normal mode where the leader is correct incurring linear message complexity, and a recovery mode when the leader is faulty and needs to be replaced incurring quadratic or higher message complexity. For example, PABC [35] achieves amortized linear message complexity in an asynchronous atomic broadcast protocol, and Zyzzyva [29] implements a linear fast-track in an SMR algorithm.

Randomization is often used to solve consensus in asynchronous networks to circumvent the seminal FLP result [22]. VABA [1] is the first multi-value asynchronous consensus algorithm that achieves an expected quadratic message complexity against a strong adaptive adversary, and other works in the asynchronous model [18, 7] achieve expected sub-quadratic communication complexity under various assumptions, but do not achieve $O(n)$ communication complexity. In the context of Byzantine SMR, HotStuff [37] specified the round synchronization conditions needed for their algorithm, and abstracted it into a module that was left unspecified. Our work provides the round synchronization they require.

Our algorithm builds on ideas presented in Cogsworth [32], but Cogsworth achieved expected linear message complexity only in the benign case, whereas in the Byzantine case its message complexity was still expected quadratic. In Cogsworth, it is enough that the first relay of a round is Byzantine to create a run with a quadratic number of messages to synchronize for that round. Since the probability that the first relay is Byzantine is constant, i.e., f/n, the overall message complexity under Byzantine failures is expected quadratic.

To reduce the expected message complexity to linear, we modified Cogsworth in a number of ways, including adding another phase to the algorithm, signing each message from a process to a relay with the relay it is intended for, and adding a “helping” mechanism to help processes “catch-up” to the latest round. By incorporating these ideas into our algorithm, we managed to bring the expected message complexity down to linear.

7 Conclusion

We presented an algorithm that reduces the expected message complexity of round synchronization to linear with an expected constant latency. Combined with algorithms like HotStuff, this yields, for the first time, Byzantine SMR with the same asymptotic performance, as round synchronization is the “bottleneck” in previous Byzantine SMR algorithms. While we achieve only expected sub-quadratic complexity, we note that achieving the same complexity in the worst-case is known to be impossible [19], and so cannot be improved.

References

1 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

2 Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 443–458. IEEE, 2014.

3 Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.

4 Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain, 2019.

5 Shai Ben-David, Allan Borodin, Richard Karp, Gabor Tardos, and Avi Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11(1):2–14, 1994.

6 Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, 1987.

7 Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. Cryptology ePrint Archive, Report 2020/851, 2020.

8 Manuel Blum. Coin flipping by telephone a protocol for solving impossible problems. *ACM SIGACT News*, 15(1):23–27, 1983.

9 Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.

10 Allan Borodin, Nathan Linial, and Michael E Saks. An optimal on-line algorithm for metrical task system. *Journal of the ACM (JACM)*, 39(4):745–763, 1992.

11 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

12 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. In *34st International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2020.

13 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *arXiv preprint arXiv:1807.04938*, 2018.

14 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

15 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

16 Miguel Castro, Barbara Liskov, et al. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

17 Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 364–369, 1986.

18 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: Sub-quadratic asynchronous byzantine agreement whp. In *34st International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2020.

19 Danny Dolev and Ruediger Reischuk. Bounds on information exchange for byzantine agreement. In *Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '82, page 132–140, New York, NY, USA, 1982. Association for Computing Machinery.

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

21 Amos Fiat, Richard Karp, Mike Luby, Lyle McGeoch, Daniel Sleator, and Neal E Young. Competitive paging algorithms. *arXiv preprint cs/0205038*, 2002.

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

23 Bryan Ford. Threshold logical clocks for asynchronous distributed coordination and consensus. *arXiv preprint arXiv:1907.07010*, 2019.

24 Eli Gafni. Round-by-round fault detectors (extended abstract) unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 143–152, 1998.

25 Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE, 2019.

26 Idit Keidar and Danny Dolev. Efficient message ordering in dynamic networks. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 68–76, 1996.

27 Idit Keidar and Alexander Shraer. Timeliness, failure-detectors, and consensus performance. In *Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 169–178, 2006.

28 Idit Keidar and Alexander Shraer. How to choose a timing model. *IEEE Transactions on Parallel and Distributed Systems*, 19(10):1367–1380, 2008.

29 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.

30 Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

31 Tal Moran, Moni Naor, and Gil Segev. An optimally fair coin toss. In *Theory of Cryptography Conference*, pages 1–18. Springer, 2009.

32 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. In *Proceedings of the Cryptoeconomic Systems Conference (CES'20)*, 2020.

33 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. *arXiv preprint arXiv:2002.07539*, 2020.

34 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.

35 HariGovind V Ramasamy and Christian Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *International Conference On Principles Of Distributed Systems*, pages 88–102. Springer, 2005.

36 Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.

37 Maofan Yin, Dahlia Malkhi, MK Reiter and, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *38th ACM symposium on Principles of Distributed Computing (PODC'19)*, 2019.