Making Byzantine Consensus Live

Manuel Bravo
IMDEA Software Institute, Madrid, Spain

Gregory Chockler
University of Surrey, UK

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Abstract

Partially synchronous Byzantine consensus protocols typically structure their execution into a sequence of *views*, each with a designated leader process. The key to guaranteeing liveness in these protocols is to ensure that all correct processes eventually overlap in a view with a correct leader for long enough to reach a decision. We propose a simple *view synchronizer* abstraction that encapsulates the corresponding functionality for Byzantine consensus protocols, thus simplifying their design. We present a formal specification of a view synchronizer and its implementation under partial synchrony, which runs in bounded space despite tolerating message loss during asynchronous periods. We show that our synchronizer specification is strong enough to guarantee liveness for single-shot versions of several well-known Byzantine consensus protocols, including HotStuff, Tendermint, PBFT and SBFT. We furthermore give precise latency bounds for these protocols when using our synchronizer. By factoring out the functionality of view synchronization we are able to specify and analyze the protocols in a uniform framework, which allows comparing them and highlights trade-offs.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Byzantine consensus, blockchain, partial synchrony, liveness

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.23

Related Version An extended version is available at `https://arxiv.org/abs/2008.04167`.

Funding This work was partially supported by an ERC Starting Grant RACCOON.

Acknowledgements We want to thank Giuliano Losa, Dahlia Malkhi, Dragos-Adrian Seredinschi, Lacramioara Astefanoaei and Eugen Zalinescu for comments that helped improve the paper.

1 Introduction

The popularity of blockchains has renewed interest in Byzantine consensus protocols, which allow a set of processes to reach an agreement on a value despite a fraction of the processes being malicious. Unlike proof-of-work or proof-of-stake protocols underlying many blockchains, classic Byzantine consensus assumes a fixed set of processes, but can in exchange provide hard guarantees on the finality of decisions. Byzantine consensus protocols are now used in blockchains with both closed membership [9, 31] and open one [15, 16, 30], in the latter case by running Byzantine consensus inside a committee elected among blockchain participants. These use cases have motivated a wave of new algorithms [15, 31, 41] that improve on classical solutions, such as DLS [27] and PBFT [20].

Designing Byzantine consensus protocols is challenging, as witnessed by a number of bugs found in recent protocols [1, 4, 7, 18]. Historically, researchers have paid more attention to safety of these protocols rather than liveness: e.g., while PBFT came with a safety proof [19], the nontrivial mechanism used to guarantee its liveness has never had one. However, achieving liveness of Byzantine consensus is no less challenging than its safety. The seminal FLP result shows that guaranteeing both properties is impossible when the network is asynchronous [28].

Hence, consensus protocols aim to guarantee safety under all circumstances and liveness only when the network is synchronous. The expected network behavior is formalized by the *partial synchrony* model [27]. In one of its more general formulations [22], the model guarantees that after some unknown *Global Stabilization Time (GST)* the system becomes synchronous, with message delays bounded by an unknown constant δ and process clocks tracking real time. Before GST, however, messages can be lost or arbitrarily delayed, and clocks at different processes can drift apart without bound. This behavior reflects real-world phenomena: in practice, the space for buffering unacknowledged messages in the communication layer is bounded, and messages will be dropped if this space overflows; also, clocks are synchronized by exchanging messages (e.g., using NTP), so network asynchrony will make clocks diverge.

Byzantine consensus protocols usually achieve liveness under partial synchrony by dividing execution into *views* (aka rounds), each with a designated leader process responsible for driving the protocol towards a decision. If a view does not reach a decision (e.g., because its leader is faulty), processes switch to the next one. To ensure liveness, the protocol needs to guarantee that all correct processes will eventually enter the same view with a correct leader and stay there long enough to complete the communication required for a decision. Achieving such *view synchronization* is nontrivial, because before GST, clocks that could measure the duration of a view can diverge, and messages that could be used to bring processes into the same view can get lost or delayed. Thus, by GST processes may end up in wildly different views, and the protocol has to bring them back together, despite any disruption caused by Byzantine processes. Some of the Byzantine consensus protocols integrate the functionality required for view synchronization with the core consensus protocol, which complicates their design [15, 20]. In contrast, both the seminal DLS work on consensus under partial synchrony [27] and some of the more recent work [3, 38, 41] suggest separating the complex functionality required for view synchronization into a distinct component – *view synchronizer*, or simply *synchronizer*. This approach allows designing Byzantine protocols modularly, with mechanisms for ensuring liveness reused among different protocols.

However, to date there has been no rigorous analysis showing which properties of a synchronizer would be sufficient for modern Byzantine consensus protocols. Furthermore, the existing implementations of synchronizer-like abstractions are either expensive or do not handle partial synchrony in its full generality. In particular, DLS [27] implements view synchronization by constructing clocks from program counters of processes. Since these counters drift apart on every step, processes need to frequently synchronize their local clocks. This results in prohibitive communication overheads and makes this solution impractical. Abraham et al. [3] address this inefficiency by assuming hardware clocks with a bounded drift, but only give a solution for a synchronous system. Finally, recent synchronizers by Naor et al. [38] only handle a simplified variant of partial synchrony which disallows clock drift and message loss before GST.

In this paper we make several contributions that address the above limitations:

- We propose a simple and precise specification of a synchronizer abstraction sufficient for single-shot consensus (§3). The specification ensures that from some point on after GST, all correct processes go through the same sequence of views, overlapping for some time in each one of them. It precisely characterizes the duration of the overlap and gives bounds on how quickly correct processes switch between views.
- We propose a synchronizer implementation, called FastSync, and rigorously prove that it satisfies our specification. FastSync handles the general version of the partial synchrony model [27], allowing for an unknown δ and – before GST – unbounded clock drift and message loss (§3.1). Despite the latter, the synchronizer runs in bounded space – a key

feature under Byzantine failures, because the absence of a bound on the required memory opens the system to denial-of-service attacks. Our synchronizer also does not use digital signatures, relying only on authenticated point-to-point links.

- We show that our synchronizer specification is strong enough to guarantee liveness under partial synchrony for single-shot versions of a number of Byzantine consensus protocols. All of these protocols can thus achieve liveness using a single synchronizer – FastSync. In the paper we consider in detail HotStuff [41] (§4.1) and its two-phase version similar to Tendermint [15] (§4.2); in an extended version [14, §B] we also analyze PBFT [20], SBFT [31] and Tendermint itself. The precise guarantees about the timing of view switches provided by our specification are key to handle such a wide range of protocols.
- We provide a precise latency analysis of FastSync, showing that it quickly converges to a synchronized view (§3.2). Building on this analysis, we prove worst-case latency bounds for the above consensus protocols when using FastSync. Our bounds consider both favorable and unfavorable conditions: if the protocol executes during a synchronous period, they determine how quickly all correct processes decide; and if the protocol starts during an asynchronous period, how quickly the processes decide after GST.
- Most of the protocols we consider were originally presented in a form optimized for solving consensus repeatedly. By specializing them to the standard single-shot consensus problem and factoring out the functionality required for view synchronization, we are able to succinctly capture their core ideas in a uniform framework. This allows us to easily compare the protocols and to shed light on trade-offs between them.

2 System Model

We assume a system of $n = 3f + 1$ processes, out of which at most f can be Byzantine, i.e., can behave arbitrarily. In the latter case the process is *faulty*; otherwise it is *correct*. We call a set Q of $2f + 1$ processes a *quorum* and write $\mathsf{quorum}(Q)$ in this case. Processes communicate using authenticated point-to-point links and, when needed, can sign messages using digital signatures. We denote by $\langle m \rangle_i$ a message m signed by process p_i. We sometimes use a cryptographic hash function $\mathsf{hash}()$, which must be collision-resistant: the probability of an adversary producing inputs m and m' such that $\mathsf{hash}(m) = \mathsf{hash}(m')$ is negligible. Processes are equipped with clocks to measure timeouts. We denote the set of time points by Time (ranged over by t) and assume that local message processing takes zero time.

We consider a generalized *partial synchrony* model [22, 27], where after some time GST message delays between correct processes are bounded by a constant δ, and both GST and δ are unknown to the protocol. Before GST messages can get arbitrarily delayed or lost (although for simplicity we assume that self-addressed messages are never lost). Assuming that both GST and δ are unknown to the protocol (as in [22]) reflects the requirements of practical systems, whose designers cannot accurately predict when network problems leading to asynchrony will stop and what the latency will be during the following synchronous period. We also assume that the processes are equipped with hardware clocks that can drift unboundedly from real time before GST, but do not drift thereafter (our results can be trivially adjusted to handle bounded clock drift after GST, but we omit this for conciseness).

3 Synchronizer Specification and Implementation

We now define a *view synchronizer* interface sufficient for single-shot Byzantine consensus, and present its specification and implementation. Let $\mathsf{View} = \{1, 2, \ldots\}$ be the set of *views*, ranged over by v; we sometimes use 0 to denote an invalid view. The job of the synchronizer is to produce notifications `new_view`(v) at each correct process, telling it to *enter* view v. A process can ensure that the synchronizer has started operating by calling a special `start`() function. We assume that each correct process eventually calls `start`().

For a consensus protocol to terminate, its processes need to stay in the same view for long enough to complete the message exchange leading to a decision. Since the message delay δ after GST is unknown to the protocol, we need to increase the view duration until it is long enough. To this end, the synchronizer is parameterized by a function defining this duration – $F : \mathsf{View} \cup \{0\} \to \mathsf{Time}$, which is monotone, satisfies $F(0) = 0$, and increases unboundedly:

$$\forall \theta.\, \exists v.\, \forall v'.\, v' \geq v \implies F(v') > \theta. \tag{1}$$

The properties on the left of Figure 1 define our synchronizer specification (ignore the properties on the right for the time being). The specification strikes a balance between usability and implementability. On one hand, it is sufficient to prove the liveness of a range of consensus protocols (as we show in §4). On the other hand, it can be efficiently implemented under partial synchrony by our FastSync synchronizer (§3.1).

Ideally, a synchronizer should ensure that all correct processes overlap in each view v for a duration determined by $F(v)$. However, achieving this before GST is impossible due to network and clock asynchrony. Therefore, we require a synchronizer to provide nontrivial guarantees only after GST and starting from some view $\mathcal{V}$. To formulate the guarantees we use the following notation. Given a view v that was entered by a correct process p_i, we denote by $E_i(v)$ the time when this happens; we let $E_{\text{first}}(v)$ and $E_{\text{last}}(v)$ denote respectively the earliest and the latest time when some correct process enters v. We let S_{first} and S_{last} be respectively the earliest and the latest time when some correct process calls `start`(), and S_k the earliest time by which k correct processes do so. Thus, a synchronizer must guarantee that views may only increase at a given process (Property 1), and ensure view synchronization starting from some view $\mathcal{V}$, entered after GST (Property 2). Starting from $\mathcal{V}$, correct processes do not skip any views (Property 3), enter each view $v \geq \mathcal{V}$ within at most d of each other (Property 4) and stay there for a determined amount of time: until $F(v)$ after the first process enters v (Property 5). Our FastSync implementation satisfies Property 4 for $d = 2\delta$. Properties 4 and 5 imply a lower bound on the overlap between the time intervals during which all correct processes execute in view v:

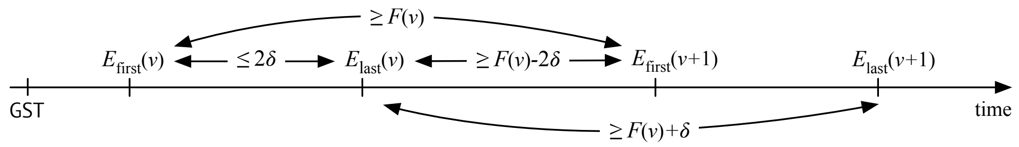
$$\forall v \geq \mathcal{V}.\, E_{\text{first}}(v+1) - E_{\text{last}}(v) \geq (E_{\text{first}}(v) + F(v)) - (E_{\text{first}}(v) + d) = F(v) - d. \tag{2}$$

Due to (1), the overlap increases unboundedly as processes keep switching views. Byzantine consensus protocols are often leader-driven, with leaders rotating round-robin across views. Hence, (2) allows us to prove their liveness by showing that there will eventually be a view with a correct leader (due to Property 3) where all correct processes will overlap for long enough. Having separate Properties 4 and 5 instead of a single property in (2) is required to prove the liveness of some protocols, e.g., two-phase HotStuff (§4.2) and Tendermint (§4.3).

3.1 FastSync: a Bounded-Space Synchronizer for Partial Synchrony

In Figure 2 we present our FastSync synchronizer, which satisfies the synchronizer specification on the left of Figure 1 for $d = 2\delta$. Despite tolerating message loss before GST, FastSync only requires bounded space; it also does not rely on digital signatures.

FastSync measures view duration using a timer `timer_view`: when the synchronizer tells the process to enter a view v, it sets the timer for the duration $F(v)$. When the timer expires, the synchronizer does not immediately move to the next view v'; instead, it



1. $\forall i, v, v'.\, (E_i(v)$ and $E_i(v')$ are defined) $\wedge\, v < v' \implies E_i(v) < E_i(v')$

2. $E_{\text{first}}(\mathcal{V}) \geq \mathsf{GST}$

3. $\forall i.\, \forall v \geq \mathcal{V}.\, p_i$ is correct $\implies p_i$ enters v

4. $\forall v \geq \mathcal{V}.\, E_{\text{last}}(v) \leq E_{\text{first}}(v) + d$

5. $\forall v \geq \mathcal{V}.\, E_{\text{first}}(v+1) \geq E_{\text{first}}(v) + F(v)$

A. $\forall v \geq \mathcal{V}.\, E_{\text{last}}(v+1) \leq E_{\text{last}}(v) + F(v) + \delta$

B. $S_{\text{first}} \geq \mathsf{GST} \wedge F(1) > 2\delta \implies$ $\mathcal{V} = 1 \wedge E_{\text{last}}(1) \leq S_{\text{last}} + \delta$

C. $F(\mathsf{GV}(\mathsf{GST} + \rho) + 1) > 2\delta \wedge S_{f+1} \leq \mathsf{GST} + \rho \implies$ $\mathcal{V} = \mathsf{GV}(\mathsf{GST} + \rho) + 1\, \wedge$ $E_{\text{last}}(\mathcal{V}) \leq \mathsf{GST} + \rho + F(\mathcal{V} - 1) + 3\delta$

Figure 1 Synchronizer properties (holding for some $\mathcal{V} \in \mathsf{View}$) and their visual illustration. Properties on the left specify the synchronizer abstraction, sufficient to ensure consensus liveness. Properties on the right give latency bounds specific to our FastSync synchronizer (§3.1). The latter satisfies Property 4 for $d = 2\delta$. The parameter ρ is the retransmission interval used by FastSync.

disseminates a special `WISH`(v') message, announcing its intention. Each process maintains an array $\mathsf{max\_views} : \{1, \ldots, n\} \to \mathsf{View} \cup \{0\}$, whose j-th entry stores the maximal view received in a `WISH` message from process p_j (initially 0, updated in line 13). Keeping track of only the maximal views allows the synchronizer to run in bounded space. The process also maintains two variables, view and view^+, derived from $\mathsf{max\_views}$ (initially 0, updated in lines 14 and 15): view^+ (respectively, view) is equal to the maximal view such that at least $f+1$ processes (respectively, $2f+1$ processes) wish to switch to a view no lower than this. The two variables monotonically increase and we always have $\mathsf{view} \leq \mathsf{view}^+$.

The process enters the view determined by the view variable (line 19) when the latter increases ($\mathsf{view} > \mathit{prev\_v}$ in line 16; we explain the extra condition later). At this point the process also resets its $\mathsf{timer\_view}$ (line 18). Thus, a process enters a view only if it receives a quorum of `WISH`es for this view or higher, and a process may be forced to switch views even if its $\mathsf{timer\_view}$ has not yet expired. The latter helps lagging processes to catch up, but poses another challenge. Byzantine processes may equivocate, sending `WISH` messages to some processes but not others. In particular, they may send `WISH`es for views $\geq v$ to some correct process, helping it to form a quorum of `WISH`es sufficient for entering v. But they may withhold the same `WISH`es from another correct process, so that it fails to form a quorum for entering v, as necessary, e.g., for Property 4. To deal with this, when a process receives a `WISH` that makes its view^+ increase, the process sends `WISH`(view^+) (line 21). By the definition of view^+, at least one correct process has wished to move to a view no lower than view^+. The `WISH`(view^+) message replaces those that may have been omitted by Byzantine processes and helps all correct processes to quickly form the necessary quorums of `WISH`es.

An additional guard on entering a view is $\mathsf{view}^+ = \mathsf{view}$ in line 16, which ensures that a process does not enter a "stale" view such that another correct process already wishes to enter a higher one. Similarly, when the timer of the current view expires (line 4), the process sends a `WISH` for the maximum of $\mathsf{view}+1$ and view^+. In other words, if $\mathsf{view} = \mathsf{view}^+$, so that the values of the two variables have not changed since the process entered the current view, then the process sends a `WISH` for the the next view ($\mathsf{view} + 1$). Otherwise, $\mathsf{view} < \mathsf{view}^+$, and the process sends a `WISH` for the higher view view^+.

```
1  function start()
2      if $\mathsf{view}^+ = 0$ then
3          send WISH(1) to all;

4  when timer_view expires
5      send WISH($\max(\mathsf{view}+1, \mathsf{view}^+)$)
         to all;

6  periodically
7      if timer_view is enabled then
8          send WISH($\mathsf{view}^+$) to all;
9      else if $\mathsf{max\_views}[i] > 0$ then
10         send WISH($\max(\mathsf{view}+1, \mathsf{view}^+)$)
             to all;

11 when received WISH($v$) from $p_j$
12     $prev\_v, prev\_v^+ \leftarrow \mathsf{view}, \mathsf{view}^+$;
13     if $v > \mathsf{max\_views}[j]$ then $\mathsf{max\_views}[j] \leftarrow v$;
14     view  $\leftarrow \max\{v \mid \exists k.\, \mathsf{max\_views}[k] = v \wedge$
                 $|\{j \mid \mathsf{max\_views}[j] \geq v\}| \geq 2f+1\}$;
15     $\mathsf{view}^+ \leftarrow \max\{v \mid \exists k.\, \mathsf{max\_views}[k] = v \wedge$
                 $|\{j \mid \mathsf{max\_views}[j] \geq v\}| \geq f+1\}$;
16     if $\mathsf{view}^+ = \mathsf{view} \wedge \mathsf{view} > prev\_v$ then
17         stop_timer(timer_view);
18         start_timer(timer_view, $F$(view));
19         trigger new_view(view);
20     if $\mathsf{view}^+ > prev\_v^+$ then
21         send WISH($\mathsf{view}^+$) to all;
```

Figure 2 The FastSync synchronizer. The periodic handler is invoked every ρ units of time.

To deal with message loss before GST, a process retransmits the highest WISH it sent every ρ units of time, according to its local clock (line 6). Depending on whether timer_view is enabled, the WISH is computed as in lines 21 or 5. Finally, the start function ensures that the synchronizer has started operating at the process by sending WISH(1), unless the process has already done so in line 21 due to receiving $f+1$ WISHes from other processes.

Discussion. FastSync requires only $O(n)$ variables for storing views. When proving its correctness, we establish that every view is entered by some correct process [14, §A, Lemma 18], and eventually, correct processes do not skip views (Property 3). These two properties limit the power of the adversary to exhaust the value space for views, similarly to [11].

The basic mechanisms we use in our synchronizer – entering views supported by $2f+1$ WISHes and relaying views supported by $f+1$ WISHes – are similar to the ones used in Bracha's algorithm for reliable Byzantine broadcast [13]. However, Bracha's algorithm only makes a step upon receiving a set of *identical* messages. Thus, its naive application to view synchronization [38, §A.2] requires unbounded space to store the views v for which the number of received copies of WISH(v) still falls below the threshold required for delivery or relay. Moreover, tolerating message loss would require a process to retain a copy of every message it has broadcast, to enable retransmissions. FastSync can be viewed as specializing the mechanisms of Bracha broadcast to take advantage of the particular semantics of WISH messages, by keeping track of only the highest WISH received from each process and by acting on sets of WISHes for non-identical views. This allows tolerating message loss before GST in bounded space and without compromising liveness, as illustrated by the following example.

We first show that, before GST, we may end up in the situation where processes are split as follows: a set P_1 of f correct processes entered v_1, a set P_2 of f correct processes entered $v_2 > v_1$, a correct process p_i entered v_2+1, and f processes are faulty. To reach this state, assume that all correct processes manage to enter view v_1 and then all messages between P_1 and $P_2 \cup \{p_i\}$ start getting lost. The f faulty processes help the processes in $P_2 \cup \{p_i\}$ to enter all views between v_1 and v_2, by providing the required WISHes (line 16), while the processes in P_1 get stuck in v_1. After the processes in $P_2 \cup \{p_i\}$ time out on v_2, they start sending WISH(v_2+1) (line 5), but all messages directed to processes other than p_i get lost, so that the processes in P_2 get stuck in v_2. The faulty processes then help p_i gather $2f+1$ messages WISH(v_2+1) and enter v_2+1 (line 16).

Assume now that GST occurs, the faulty processes go silent and the correct processes time out on the views they are in. Thus, the f processes in P_1 send WISH(v_1+1), the f

processes in P_2 send $\texttt{WISH}(v_2+1)$, and p_i sends $\texttt{WISH}(v_2+2)$ (line 10). The processes in P_1 eventually receive the $\texttt{WISH}$es from $P_2 \cup \{p_i\}$, so that they set $\mathsf{view}^+ = v_2+1$ and send $\texttt{WISH}(v_2+1)$ (line 21). Note that here processes act on $f+1$ mismatching $\texttt{WISH}$es, unlike in Bracha broadcast. Eventually, the processes in $P_1 \cup P_2$ receive $2f$ copies of $\texttt{WISH}(v_2+1)$ and one $\texttt{WISH}(v_2+2)$, which causes them to set $\mathsf{view} = v_2+1$ and enter v_2+1 (line 16). Note that here processes act on $2f+1$ mismatching $\texttt{WISH}$es, again unlike in Bracha broadcast. Finally, the processes $P_1 \cup P_2$ time out and send $\texttt{WISH}(v_2+2)$ (line 5), which allows all correct processes to enter v_2+2. Acting on sets of mismatching $\texttt{WISH}$es is crucial for liveness in this example: if processes only accepted matching sets, like in Bracha broadcast, message loss before GST would cause them to get stuck, and they would never converge to the same view.

3.2 Correctness and Latency Bounds of FastSync

As we demonstrate shortly, the synchronizer specification given by Properties 1-5 in Figure 1 serves to prove that consensus *eventually* reaches a decision. However, FastSync also satisfies some additional properties that allow us to quantify *how quickly* this happens under both favorable and unfavorable conditions. We list these properties on the right of Figure 1.

▶ **Theorem 1.** FastSync *satisfies all properties in Figure 1 for* $d = 2\delta$.

Due to space constraints, we defer the proof to [14, §A]. Property A allows us to quantify the cost of switching between several views (e.g., due to faulty leaders). This is formalized by the following proposition, easily proved using Property A by induction on v'.

▶ **Proposition 2.** $\forall v, v'.\, \mathcal{V} \le v \le v' \implies E_{\text{last}}(v') \le E_{\text{last}}(v) + \sum_{k=v}^{v'-1}(F(k)+\delta)$.

Property B guarantees that, when the synchronizer starts after GST ($S_{\text{first}} \ge \mathsf{GST}$) and the initial timeout is long enough ($F(1) > 2\delta$), processes synchronize in the very first view ($\mathcal{V} = 1$) and enter it within δ of the last correct process calling $\texttt{start}()$.

Let the *global view* at time t, denoted $\mathsf{GV}(t)$, be the maximum view entered by a correct process at or before t, or 0 if no view was entered by a correct process. Property C quantifies the latency of view synchronization in a more general case when the synchronizer may be started before GST. The property depends on the interval ρ at which the synchronizer periodically retransmits its internal messages to deal with possible message loss. The property considers the highest view $\mathsf{GV}(\mathsf{GST}+\rho)$ a correct process has at time $\mathsf{GST}+\rho$ and ensures that all correct processes synchronize in the immediately following view within at most $\rho + F(\mathcal{V}-1) + 3\delta$ after GST. This is guaranteed under an assumption that the timeout of this view exceeds 2δ and $f+1$ correct processes call $\texttt{start}()$ early enough. Since GST can be arbitrary, in principle, so can be the view $\mathcal{V}$ and, thus due to (1), the timeout $F(\mathcal{V}-1)$. However, practical implementations usually stop increasing timeouts when they exceed a reasonable value. Hence, Property C guarantees that to reach $\mathcal{V}$, processes need to wait for at most a single maximal timeout.

4 Liveness and Latency of Byzantine Consensus Protocols

We show that our synchronizer abstraction allows ensuring liveness and establishing latency bounds for several consensus protocols. The protocols solve a variant of Byzantine consensus problem that relies on an application-specific $\mathsf{valid}()$ predicate to indicate whether a value is valid [17, 24]. In the context of blockchain systems a value represents a block, which may be invalid if it does not include correct signatures authorizing its transactions. Assuming that each correct process proposes a valid value, each of them has to decide on a value so that:

- **Agreement.** No two correct processes decide on different values.
- **Validity.** A correct process decides on a valid value, i.e., satisfying valid().
- **Termination.** Every correct process eventually decides on a value.

4.1 Single-Shot HotStuff

We first consider the HotStuff protocol [41], underlying the upcoming Libra cryptocurrency [2]. The protocol was originally presented as solving an inherently multi-shot problem, agreeing on a hash-chain of blocks. In Figure 3 we present its single-shot version that concisely expresses the key idea and allows comparing the protocol with others. For brevity, we eschew the use of threshold signatures, which makes the communication complexity of a leader change $O(n^2)$ rather than $O(n)$, like in the original HotStuff. This complexity is still better than that of PBFT, which is $O(n^3)$. We handle linear versions of the protocols we consider in [14, §C]. HotStuff delegated view synchronization to a separate component [41], but did not provide its practical implementation or analyze how view synchronization affects the protocol latency. We show that our single-shot version of HotStuff is live when used with a synchronizer satisfying the specification in §3 and give precise bounds on its latency. We also show that the protocol requires only bounded space when using our synchronizer FastSync.

The protocol in Figure 3 works in a succession of views produced by the synchronizer. Each view v has a fixed leader $\mathsf{leader}(v) = p_{((v-1) \bmod n)+1}$ that is responsible for proposing a value to the other processes, which vote on the proposal. A correct leader needs to choose its proposal carefully so that, if a value was decided in a previous view, the leader will propose the same value. To enable the leader to do this, when a process receives a notification to move to a view v (line 1), it sends a NEWLEADER message to the leader of v with information about the latest value it accepted in a previous view (as described in the following). The process also stores the view v in a variable curr_view, and sets a flag voted to FALSE, to record that it has not yet received any proposal from the leader in the current view. The leader computes its proposal (as described in the following) based on a quorum of NEWLEADER messages (line 5) and sends the proposal, along with some supporting information, in a PROPOSE message to all processes (for uniformity, including itself).

The leader's proposal is processed in three phases. A process receiving a proposal x from the leader of its view v (line 11) first checks that voted is FALSE, so that it has not yet accepted a proposal in v. It also checks that x satisfies a SafeProposal predicate (explained later), which ensures that a faulty leader cannot reverse decisions reached in previous views. The process then sets voted to TRUE and stores x in curr_val.

Since a faulty leader may send different proposals to different processes, the process next communicates with others to check that they received the same proposal. To this end, the process disseminates a PREPARED message with the hash of the proposal it received. The process then waits until it gathers a set C of PREPARED messages from a quorum with a hash matching the proposal (line 16); we call this set of messages a *prepared certificate* for the value and check it using the prepared predicate. The process stores the proposal in prepared_val, the view in which it formed the prepared certificate in prepared_view, and the certificate itself in cert. At this point we say that the process *prepared* the value. Since a certificate consists of at least $2f + 1$ PREPARED messages and there are $3f + 1$ replicas in total, it is impossible to prepare different values in the same view: this would require some correct process to send two PREPARED messages with different values in the same view, which is impossible due to

```
1  **upon** new_view($v$)
2      curr_view $\leftarrow v$;
3      voted $\leftarrow$ FALSE;
4      **send** $\langle$NEWLEADER(curr_view, prepared_view,
           prepared_val, cert)$\rangle_i$ **to** leader(curr_view);

5  **when received** $\{\langle$NEWLEADER($v$, $view_j$, $val_j$,
       $cert_j$)$\rangle_j \mid p_j \in Q\} = M$ **for a quorum** $Q$
6      **pre:** curr_view $= v \wedge p_i =$ leader($v$) $\wedge$
           ($\forall m \in M.$ ValidNewLeader($m$));
7      **if** $\exists j.\, view_j = \max\{view_k \mid p_k \in Q\} \neq 0$ **then**
8          **send** $\langle$PROPOSE($v$, $val_j$, $cert_j$)$\rangle_i$ **to all**;
9      **else**
10         **send** $\langle$PROPOSE($v$, myval(), $\perp$)$\rangle_i$ **to all**;

11 **when received** $\langle$PROPOSE($v, x, \_$)$\rangle_j = m$
12     **pre:** curr_view $= v \wedge$ voted $=$ FALSE $\wedge$
           SafeProposal($m$);
13     curr_val $\leftarrow x$;
14     voted $\leftarrow$ TRUE;
15     **send** $\langle$PREPARED($v$, hash(curr_val))$\rangle_i$ **to all**;

16 **when received** $\{\langle$PREPARED($v, h$)$\rangle_j \mid$
       $p_j \in Q\} = C$ **for a quorum** $Q$
17     **pre:** curr_view $= v \wedge$ voted $=$ TRUE
           $\wedge$ hash(curr_val) $= h$;
18     prepared_val $\leftarrow$ curr_val;
19     prepared_view $\leftarrow$ curr_view;
20     cert $\leftarrow C$;
21     **send** $\langle$PRECOMMITTED($v, h$)$\rangle_i$ **to all**;

22 **when received** $\{\langle$PRECOMMITTED($v, h$)$\rangle_j$
       $\mid p_j \in Q\}$ **for a quorum** $Q$
23     **pre:** curr_view $=$ prepared_view $= v \wedge$
           hash(curr_val) $= h$;
24     locked_view $\leftarrow$ prepared_view;
25     **send** $\langle$COMMITTED($v, h$)$\rangle_i$ **to all**;

26 **when received** $\{\langle$COMMITTED($v, h$)$\rangle_j \mid$
       $p_j \in Q\}$ **for a quorum** $Q$
27     **pre:** curr_view $=$ locked_view $= v \wedge$
           hash(curr_val) $= h$;
28     decide(curr_val);
```

$$\mathsf{prepared}(C, v, h) \iff \exists Q.\, \mathsf{quorum}(Q) \wedge C = \{\langle \mathtt{PREPARED}(v, h)\rangle_j \mid p_j \in Q\}$$

$$\mathsf{ValidNewLeader}(\langle \mathtt{NEWLEADER}(v', v, x, C)\rangle_{\_}) \iff v < v' \wedge (v \neq 0 \implies \mathsf{prepared}(C, v, \mathsf{hash}(x)))$$

$$\mathsf{SafeProposal}(\langle \mathtt{PROPOSE}(v, x, C)\rangle_i) \iff p_i = \mathsf{leader}(v) \wedge \mathsf{valid}(x) \wedge$$
$$(\mathsf{locked\_view} \neq 0 \implies x = \mathsf{prepared\_val} \vee (\exists v'.\, v > v' > \mathsf{locked\_view} \wedge \mathsf{prepared}(C, v', \mathsf{hash}(x))))$$

Figure 3 Single-shot HotStuff. All variables storing views are initially set to 0 and others to $\perp$.

the check on the voted flag in line 12. Formally, let us write wf(C) (for *well-formed*) if the set of correctly signed messages C have been sent in the execution of the protocol.

▶ **Proposition 3.** $\forall v, C, C', x, x'.\, \mathsf{prepared}(C, v, \mathsf{hash}(x)) \wedge \mathsf{prepared}(C', v, \mathsf{hash}(x')) \wedge \mathsf{wf}(C) \wedge \mathsf{wf}(C') \implies x = x'$.

Preparing a value is a prerequisite for deciding on it. Hence, by Proposition 3 a prepared certificate for a value x and a view v guarantees that x is the only value that can be possibly decided in v. For this reason, it is this certificate, together with the corresponding value and view, that the process sends upon a view change to the new leader in a NEWLEADER message (line 4). The leader makes its proposal based on a quorum of NEWLEADER messages with prepared certificates formed in lower views than the one it is in (line 5), as checked by ValidNewLeader. Similarly to Paxos [34] and PBFT [20], the leader selects as its proposal the value prepared in the highest view, or, if there are no such values, its own proposal given by myval(). In the former case, the leader sends the corresponding certificate in its PROPOSE message, to justify its choice; in the latter case this is replaced by $\perp$.

Once a process prepares a value x, it participates in the next message exchange: it disseminates a PRECOMMITTED message with the hash of the value and waits until it gathers a quorum of PRECOMMITTED messages matching the prepared value (line 22). This ensures that at least $f + 1$ correct processes have prepared the value x. Since the leader of the next view will gather prepared commands from at least $2f + 1$ processes, at least one correct process will tell the leader about the value x, and thus the leader will be aware of this value as a potential decision in the current view.

Having gathered a quorum of PRECOMMITTED messages for a value, the process becomes *locked* on this value, which is recorded by setting a special variable locked_view to the current view. From this point on, the process will not accept a proposal of a different value from a leader of a future view, unless the leader can convince the process that no decision was reached in the current view. This is ensured by the SafeProposal check the process does on a PROPOSE message from a leader (line 12). This checks that the value is valid and that, if the process has previously locked on a value, then either the leader proposes the same value, or its proposal is justified by a prepared certificate from a higher view than the lock. In the latter case the process can be sure that no decision was reached in the view it is locked on.

Having locked a value, the process participates in the final message exchange: it disseminates a COMMITTED message with the hash of the value and waits until it gathers a quorum of matching COMMITTED messages for the locked value (line 26). Once this happens, the process decides on this value. Gathering a quorum of COMMITTED messages on a value x ensures that at least $f+1$ correct processes are locked on the same value. This guarantees that a leader in a future view cannot get processes to decide on a different value: this would require $2f+1$ processes to accept the leader's proposal; but at least one correct process out of these would be locked on x and would refuse to accept a different value due to the SafeProposal check. Thus, while the exchange of PRECOMMITTED messages ensures that a future correct leader will be aware of the value being decided and will be able to make a proposal passing SafeProposal checks (liveness), the exchange of COMMITTED ensures that a faulty leader cannot revert the decision (safety).

Since processes transition through increasing views (Property 1 in Figure 1), we get

▶ **Proposition 4.** *The variables* locked_view, prepared_view *and* curr_view *at a correct process never decrease and we always have* locked_view $\leq$ prepared_view $\leq$ curr_view.

Note that, when a process enters view 1, it trivially knows that no decision could have been reached in prior views. Hence, the leader of view 1 can send its proposal immediately, without waiting to receive a quorum of NEWLEADER messages (line 10), and processes can avoid sending these messages to this leader. For brevity, we omit this optimization from the pseudocode, even though we take it into account in our latency analysis.

Since the synchronizer is not guaranteed to switch processes between views all at the same time, a process in a view v may receive a message from a higher view $v' > v$, which needs to be stored in case the process finally switches to v'. If implemented naively, this would require a process to store unboundedly many messages. Instead, we allow a process to store, for each message type and sender, only the message of this type received from this sender that has the highest view. As we show below (Theorem 5), this does not violate liveness. Thus, assuming consensus proposals of bounded size, the protocol Figure 3 runs in bounded space, and so does the overall consensus protocol with the FastSync synchronizer.

We defer the proof that the protocol satisfies Validity and Agreement to [14, §B.1] and focus on our core contribution: proving its liveness and analyzing its latency.

Protocol liveness. Assume that the protocol is used with a synchronizer satisfying Properties 1-5 on the left of Figure 1; to simplify the following latency analysis, we assume $d = 2\delta$, as for FastSync. The next theorem states requirements on a view sufficient for the protocol to reach a decision and quantifies the resulting latency.

▶ **Theorem 5.** *Let $v \geq \mathcal{V}$ be a view such that $F(v) > 7\delta$ and* leader(v) *is correct. Then in single-shot HotStuff all correct processes decide in view v by $E_{\text{last}}(v) + 5\delta$.*

Proof. By Property 2 we have $E_{\text{first}}(v) \geq \mathsf{GST}$, so that all messages sent by correct processes after $E_{\text{first}}(v)$ get delivered to all correct processes within δ. Once a correct process enters v, it sends its `NEWLEADER` message, so that $\mathsf{leader}(v)$ will receive a quorum of such messages by $E_{\text{last}}(v)+\delta$. When this happens, the leader will send its proposal in a `PROPOSE` message, which correct processes will receive by $E_{\text{last}}(v) + 2\delta$. If they deem the proposal safe, it takes them at most 3δ to exchange the sequence of `PREPARED`, `PRECOMMITTED` and `COMMITTED` messages. By (2), all correct processes will stay in v until at least $E_{\text{last}}(v) + (F(v) - d) > E_{\text{last}}(v) + 5\delta$, and thus will not send a message with a view $> v$ until this time. Thus, none of none of the above messages will be discarded at correct processes before this time, and assuming the safety checks pass, the sequence of message exchanges will lead to decisions by $E_{\text{last}}(v) + 5\delta$.

It remains to show that the proposal $\mathsf{leader}(v)$ makes in view v (line 5) will satisfy $\mathsf{SafeProposal}$ at all correct processes (line 11). It is easy to show that the proposal satisfies valid, so we now need to prove the last conjunct of $\mathsf{SafeProposal}$. This trivially holds if no correct process is locked on a value when receiving the `PROPOSE` message from the leader.

We now consider the case when some correct process is locked on a value when receiving the `PROPOSE` message, and let p_i be a process that is locked on the highest view among correct processes. Let $x = p_i.\mathsf{prepared\_val}$ be the value locked and $v_0 = p_i.\mathsf{locked\_view} < v$ be the corresponding view. Since p_i locked x at v_0, it must have previously received messages $\texttt{PRECOMMITTED}(v_0, \mathsf{hash}(x))$ from a quorum of processes (line 22), at least $f + 1$ of which have to be correct. The latter processes must have prepared the value x at view v_0 (line 16). By Proposition 4, when each of these $f + 1$ correct processes enters view v, it has $\mathsf{prepared\_view} \geq v_0$ and thus sends the corresponding value and its prepared certificate in the $\texttt{NEWLEADER}(v, \dots)$ message to $\mathsf{leader}(v)$. The leader is guaranteed to receive at least one of these messages before making a proposal, since it only does this after receiving at least $2f + 1$ `NEWLEADER` messages (line 5). Hence, the leader proposes a value x' with a prepared certificate formed at some view $v' \geq v_0$ no lower than any view that a correct process is locked on when receiving the leader's proposal. Furthermore, if $v' = v_0$, then by Proposition 3 we have that $x' = x$ and x is the only value that can be locked by a correct process at v_0. Hence, the leader's proposal will satisfy $\mathsf{SafeProposal}$ at each correct process. ◀

Since by Property 3 correct processes enter every view starting from $\mathcal{V}$ and, by the definition of $\mathsf{leader}()$, leaders rotate round-robin, we are always guaranteed to encounter a correct leader after at most f view changes. Then Theorem 5 implies that the protocol is live when using a timeout function F that grows without bound.

▶ **Corollary 6.** *Let F be such that (1) holds. Then in single-shot HotStuff all correct processes eventually decide.*

Protocol latency. When single-shot HotStuff is used with the FastSync synchronizer, rather than an arbitrary one, we can use Properties A-C on the right of Figure 1 to bound how quickly the protocol reaches a decision after GST. To this end, we combine Theorem 5 with Property C, which bounds the latency of view synchronization, and Proposition 2, which bounds the latency of going through up to f views with faulty leaders.

▶ **Corollary 7.** *Let $v = \mathsf{GV}(\mathsf{GST} + \rho) + 1$ and assume that $F(v) > 7\delta$ and $S_{f+1} \leq \mathsf{GST} + \rho$. Then in single-shot HotStuff all correct processes decide by $\mathsf{GST} + \rho + \sum_{k=v-1}^{v+f-1}(F(k) + \delta) + 7\delta$.*

We can also quantify the latency of the protocol under favorable conditions, when it is started after GST. In this we rely on Property B, which gives conditions under which processes synchronize in view 1. The following corollary of Theorem 5 exploits this property to bound the latency of HotStuff when it is started after GST and the initial timeout is set appropriately, but the protocol may still go through a sequence of up to f faulty leaders. The summation in the bound (coming from Proposition 2) quantifies the overhead in the latter case.

▶ **Corollary 8.** *Assume that $S_{\text{first}} \geq \mathsf{GST}$ and $F(1) > 7\delta$. Then in single-shot HotStuff all correct processes decide no later than $S_{\text{last}} + \sum_{k=1}^{f}(F(k) + \delta) + 6\delta$.*

Finally, the next corollary bounds the latency when additionally the leader of view 1 is correct, in which case the protocol can benefit from the optimized execution of this view noted earlier. The corollary follows from Property B and an easy strengthening of Theorem 5 for the special case of $v = \mathcal{V} = 1$.

▶ **Corollary 9.** *Assume that $S_{\text{first}} \geq \mathsf{GST}$, $F(1) > 6\delta$, and $\mathsf{leader}(1)$ is correct. Then in single-shot HotStuff all correct processes decide no later than $S_{\text{last}} + 5\delta$.*

4.2 Two-Phase HotStuff

We next consider a *two-phase* variant of HotStuff [41], which processes the leader's proposals in two phases instead of three. In exchange, it uses timeouts not just for view synchronization, but also in the core consensus protocol to delimit different stages of a single view. This demonstrates that our synchronizer specification is strong enough to deal with interactions between the timeouts in different parts of the overall protocol. When used with our FastSync synchronizer, the protocol furthermore requires only bounded space. Two-phase HotStuff is similar to Tendermint [15] and Casper [16], which use timeouts for the same purposes. We chose this protocol for conciseness of presentation, but in [14, §B.5] we also present a variant of the original Tendermint consensus based on our synchronizer (see §4.3).

Due to space constraints, we describe the changes to the protocol in Figure 3 required to get its two-phase version informally and defer the pseudocode to [14, §B.2]. In two-phase HotStuff, a process handles a proposal from the leader in the same way as in the three-phase one, by sending a `PREPARED` message (line 11 in Figure 3). Upon assembling a quorum of matching `PREPARED` messages (line 16), the process updates its variables as per lines 18-20, but in addition immediately becomes locked on the prepared value prepared_val, without exchanging `PRECOMMITTED` messages: the process assigns locked_view to the current view and sends a `COMMITTED` message with the hash of the value. As before, assembling a quorum of such messages causes the process to decide on the value (line 26). Upon entering a new view (line 1), a process sends to the leader a `NEWLEADER` message with the information about the last value it prepared (and therefore locked, line 4). The leader chooses its proposal in the same way as in three-phase HotStuff (line 5).

The two-phase version of HotStuff is safe for the same reasons as the three-phase one: the exchange of `PRECOMMITTED` messages, omitted from the current protocol, is only needed for liveness, not safety. However, ensuring liveness in two-phase HotStuff requires a different mechanism: since a correct process p_i gets locked on a value immediately after preparing it, gathering prepared values from an arbitrary quorum of processes is not enough for the leader to ensure it will make a proposal that will pass the SafeProposal check at p_i: the quorum may well exclude this process. To solve this problem, the leader waits before making a proposal so that eventually in some view it will receive `NEWLEADER` messages from *all correct processes*. This ensures the leader will eventually make a proposal that will pass the SafeProposal checks at all of them. In more detail, when a process enters a view where it is the leader, it sets a special timer timer_newleader for the duration determined by a function F_p. The leader makes a proposal by executing the handler in line 5 only after the timer expires.

For the leader to make an acceptable proposal, the duration of timer_newleader needs to be long enough for all NEWLEADER messages for this view from correct processes to reach the leader. For the protocol to decide, after timer_newleader expires, processes also need to stay in the view long enough to complete the necessary message exchanges. The following theorem characterizes these requirements formally, again assuming $d = 2\delta$ in Property 4. Note that in the proof of the theorem we rely on the guarantees about the timing of correct processes entering a view (Property 4) to show that timer_newleader fulfills its intended function.

▶ **Theorem 10.** *Let $v \geq \mathcal{V}$ be a view such that $F_p(v) > 3\delta$, $F(v) - F_p(v) > 5\delta$ and* leader(v) *is correct. Then in two-phase HotStuff all correct processes decide at v by $E_{\text{last}}(v) + F_p(v) + 3\delta$.*

Proof. Once a correct process enters v, it sends its NEWLEADER message, so that leader(v) is guaranteed to receive such messages from all correct processes by $E_{\text{last}}(v) + \delta$. By Property 4, the leader enters v by $E_{\text{last}}(v) - 2\delta$ at the earliest. Since the leader starts its timer_newleader when it enters v and $F_p(v) > 3\delta$, timer_newleader can only expire after $E_{\text{last}}(v) + \delta$. Thus, the leader is guaranteed to receive NEWLEADER messages from all correct processes before timer_newleader expires. When timer_newleader expires, which happens no later than $E_{\text{last}}(v) + F_p(v)$, the leader will send its proposal in a PROPOSE message, which correct processes will receive by $E_{\text{last}}(v) + F_p(v) + \delta$. If they deem the proposal safe, it takes them at most 2δ to exchange the sequence of PREPARED and COMMITTED messages leading to decisions. By (2), all correct processes will stay in v until at least $E_{\text{last}}(v) + (F(v) - d) > E_{\text{first}}(v) + F_p(v) + 3\delta$. By then the above sequence of message exchanges will complete, and all correct processes will decide.

It remains to show that the proposal leader(v) makes in view v will satisfy SafeProposal at all correct processes. It is easy to show that this proposal is valid, so we now need to prove the last conjunct of SafeProposal. This trivially holds if no correct process is locked on a value when receiving the PROPOSE message from the leader. We now consider the case when some correct process is locked on a value when receiving the PROPOSE message, and let p_i be a process that is locked on the highest view among correct processes. Let $x = p_i$.prepared_val be the value locked and $v_0 = p_i$.locked_view $< v$ be the corresponding view. Since leader(v) receives all of the NEWLEADER messages sent by correct processes before making its proposal, it proposes a value x' with a prepared certificate formed at some view $v' \geq v_0$. Also, if $v' = v_0$, then by Proposition 3, $x' = x$ and x is the only value that can be locked by a correct process at v_0. Hence, the leader's proposal will satisfy SafeProposal at each correct process. ◀

Since leaders rotate round-robin, Theorem 10 implies that the protocol is live, provided the functions F and F_p, *as well as the difference between them*, grow without bound. This can be satisfied, e.g., by letting $F(v) = 2v$ and $F_p(v) = v$.

▶ **Corollary 11.** *Let F and F_p be such that (1) holds and $\forall \theta. \exists v. \forall v'. v' \geq v \implies F(v') - F_p(v') > \theta$. Then in two-phase HotStuff all correct processes eventually decide.*

Protocol latency. Similarly to §4.1, when the protocol is used with the FastSync synchronizer, we can quantify its latency in both unfavorable scenarios (when starting before GST) and favorable scenarios (when starting after GST). The first corollary of Theorem 10 below uses Property C and Proposition 2, and the following two corollaries, Property B.

▶ **Corollary 12.** *Let $v = \mathsf{GV}(\mathsf{GST} + \rho) + 1$ and assume that $S_{f+1} \leq \mathsf{GST} + \rho$, $F_p(v) > 3\delta$ and $F(v) - F_p(v) > 5\delta$. Then in two-phase HotStuff all correct processes decide no later than $\mathsf{GST} + \rho + \sum_{k=v-1}^{v+f-1}(F(k) + \delta) + F_p(v + f) + 5\delta$.*

▶ **Corollary 13.** *Assume that* $S_{\text{first}} \geq \mathsf{GST}$*,* $F_p(1) > 3\delta$ *and* $F(1) - F_p(1) > 5\delta$*. Then in two-phase HotStuff all correct processes decide no later than* $S_{\text{last}} + \sum_{k=1}^{f}(F(k)+\delta)+F_p(f+1)+4\delta$*.*

▶ **Corollary 14.** *Assume that* $S_{\text{first}} \geq \mathsf{GST}$*,* $F(1) > 5\delta$ *and* $\mathsf{leader}(1)$ *is correct. Then in two-phase HotStuff all correct processes decide no later than* $S_{\text{last}} + 4\delta$*.*

Like in §4.1, the last corollary takes into account the optimized execution of view 1. The above latency bounds allow us to compare the two-phase and three-phase versions of HotStuff (§4.1). In the ideal case when the timeouts are set optimally and the leader of view 1 is correct, two-phase HotStuff has a lower latency than three-phase one: 4δ in Corollary 14 vs 5δ in Corollary 9. When the initial leader is faulty, both protocols incur the overhead of switching through several views until they encounter a correct leader (Corollaries 13 and 8). In this case, the latency of deciding in the first view with a correct leader is at most 6δ for three-phase HotStuff and $F_p(f+1)+4\delta$ for two-phase one. Even when $F_p(f+1)$ is the optimal 3δ, the two-phase HotStuff bound yields 7δ – a higher latency than for three-phase HotStuff. The latency bounds for the case of starting before GST relate similarly (Corollaries 12 and 7). The higher latency of two-phase HotStuff in these cases are caused by the inclusion of the timeout $F_p(f+1)$, which reflects the lack of “optimistic responsiveness” of this protocol [41].

4.3 Single-Shot PBFT, SBFT and Tendermint

Using our synchronizer specification, we have also proved the correctness and analyzed the latency of single-shot versions of PBFT [20], SBFT [31] and Tendermint [15], thus demonstrating the wide applicability of the specification. Due to space constraints we defer the details to [14, §B]. Our analysis of PBFT is similar to that of HotStuff. SBFT is a recent improvement of PBFT that adds a fast path for cases when all processes are correct, and our analysis quantifies the latency of both paths.

Tendermint is similar to two-phase HotStuff; in particular, it also uses timeouts both for view synchronization and to delimit different stages of a single view. However, the protocol never sends messages with certificates, and thus, like FastSync, does not need digital signatures. Tendermint integrates the functionality required for view synchronization with the core consensus protocol, breaking its control flow in multiple places. We consider its variant that delegates this functionality to the synchronizer, thus simplifying the protocol. Our analysis of the resulting protocol is similar to the one of two-phase HotStuff in §4.2. Apart from deriving latency bounds for the protocol, our analysis exploits the synchronizer specification to give a proof of its liveness that is more rigorous than the existing ones [8, 15], which lacked a detailed correctness argument for the view synchronization mechanism used in the protocol.

5 Related Work

Most Byzantine consensus protocols are based on the concept of views (aka rounds), and thus include a mechanism for view synchronization. This mechanism is typically integrated with the core consensus protocol, which complicates the design [15, 20, 31]. Subtle view synchronization mechanisms have often come without a proof of liveness (e.g., PBFT [19]) or had liveness bugs (e.g., Tendermint [7] and Casper [1]). Furthermore, liveness proofs have not usually given concrete bounds on the latency of reaching a decision (exceptions are [6, 36]).

Several papers suggested separating the functionality of view synchronization into a distinct component, starting with the seminal DLS paper on consensus under partial synchrony [27]. DLS specified the guarantees provided by view synchronization indirectly, by proving that its implementation simulated an abstract computational model with a built-in notion of rounds. Unlike us, DLS did not give a specification determining how long processes stay in a round and how quickly they switch between rounds; as we have demonstrated, such properties are needed to reason about modern Byzantine consensus protocols. DLS implemented rounds using a distributed protocol that synchronizes process-local clocks obtained by counting state transitions of each process. This protocol has to synchronize local clocks on every step of the consensus algorithm, which results in prohibitive communication overheads and makes this solution impractical.

Abraham et al. [3] build upon ideas from fault-tolerant clock synchronization [25, 40] to implement view synchronization assuming that processes have access to hardware clocks with bounded drift. But this work only gives a solution for a synchronous system. Our FastSync synchronizer also assumes hardware clocks but removes the assumption of bounded drift before GST, thus making them compatible with partial synchrony. We note that, although the problems of clock and view synchronization are different, they are closely related at the algorithmic level. We therefore believe that our view synchronization techniques can in the future be adapted to obtain an efficient partially synchronous clock synchronization protocol.

The HotStuff protocol [41] delegated the functionality of view synchronization to a separate component, called a pacemaker. But it did not provide a formal specification of this component or a practical implementation. To address this, Naor et al. have recently formalized view synchronization as a separate problem [38, 39]. Unlike us, they did not provide a comprehensive study of the applicability of their specifications to a wide range of modern Byzantine consensus protocols. In particular, their specifications do not expose bounds on how quickly processes switch views (Property 4 in Figure 1), which are necessary for protocols such as two-phase HotStuff (§4.2) and Tendermint (§4.3).

Naor et al. also proposed synchronizer implementations in a simplified variant of partial synchrony where δ is known a priori, and messages sent before GST are guaranteed to arrive by $\mathsf{GST} + \delta$ [38, 39]. These implementations focus on optimizing communication complexity, making it linear in best-case scenarios [38] or in expectation [39]. They achieve linearity by relying on digital signatures (more precisely, threshold signatures), which FastSync eschews. Unlike FastSync, they also require unbounded space (for the reasons explained in §3.1). Finally, we give exact latency bounds for FastSync under both favorable and unfavorable conditions whereas [38,39] only provide expected latency analysis. It is interesting to investigate whether the benefits of the two approaches can be combined to tolerate message loss before GST with both bounded space and a low communication complexity.

LibraBFT [2] extends HotStuff with a view synchronization mechanism, integrated with the core protocol; the protocol assumes reliable channels. LibraBFT is optimized to solve repeated consensus, whereas in this paper we focus on single-shot one. We leave investigating synchronizer abstractions optimized for the multi-shot case to future work.

The original idea of using synchronizers to simulate a round-based synchronous system on top of an asynchronous one is due to Awerbuch [10]. This work however, did not consider failures. Augmented round models to systematically study properties of distributed consensus under various failure and environment assumptions were proposed in [12, 23, 29, 33]. These papers however, do not deal with implementing the proposed models under partial synchrony. Upper bounds for deciding after GST in round-based crash fault-tolerant consensus algorithms were studied in [5, 26]. While we derive similar bounds for Byzantine failures, it remains open if these are optimal or can be further improved. Failure detectors [21, 22], which abstract away the timeliness guarantees of the environment, have been extensively used for developing and analyzing consensus algorithms [22, 37] in the presence of benign failures. However, since capturing all possible faulty behaviors is algorithm-specific, the classical notion of a failure detector does not naturally generalize to Byzantine settings. As a result, the existing work on Byzantine failure detectors either limits the types of failures being addressed (e.g., [35]), or focuses on other means (such as accountability [32]) to mitigate faulty behavior.

References

1 Incorrect by construction-CBC Casper isn't live. https://pyrofex.io/wp-content/uploads/2018/12/Incorrect-By-Construction.pdf.

2 State machine replication in the Libra blockchain. https://developers.libra.org/docs/assets/papers/libra-consensus-state-machine-replication-in-the-libra-blockchain.pdf.

3 Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous Byzantine agreement with expected $O(1)$ rounds, expected $O(n^2)$ communication, and optimal resilience. In *Conference on Financial Cryptography and Data Security (FC)*, 2019.

4 Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. Revisiting fast practical Byzantine fault tolerance. *arXiv*, abs/1712.01367, 2017. `arXiv:1712.01367`.

5 Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. How to solve consensus in the smallest window of synchrony. In *Symposium on Distributed Computing (DISC)*, 2008.

6 Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Trans. Dependable Sec. Comput.*, 8(4):564–577, 2011.

7 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Correctness of Tendermint-core blockchains. In *Conference on Principles of Distributed Systems (OPODIS)*, 2018.

8 Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Dissecting Tendermint. In *Conference on Networked Systems (NETYS)*, 2019.

9 Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *European Conference on Computer Systems (EuroSys)*, 2018.

10 Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985.

11 Rida A. Bazzi and Yin Ding. Non-skipping timestamps for Byzantine data storage systems. In *Symposium on Distributed Computing (DISC)*, 2004.

12 Martin Biely, Josef Widder, Bernadette Charron-Bost, Antoine Gaillard, Martin Hutle, and André Schiper. Tolerating corrupted communication. In *Symposium on Principles of Distributed Computing (PODC)*, 2007.

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

14 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making Byzantine consensus live (extended version). *arXiv*, abs/2008.04167, 2020. `arXiv:2008.04167`.

15 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *arXiv*, abs/1807.04938, 2018. `arXiv:1807.04938`.

16 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv*, abs/1710.09437, 2017. `arXiv:1710.09437`.

17 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *International Cryptology Conference (CRYPTO)*, 2001.

18 Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild (keynote talk). In *Symposium on Distributed Computing (DISC)*, 2017.

19 Miguel Castro. *Practical Byzantine Fault Tolerance.* PhD thesis, Massachusetts Institute of Technology, 2001.

20 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

21 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

22 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

23 Bernadette Charron-Bost and André Schiper. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Comput.*, 22(1):49–71, 2009.

24 Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: efficient leaderless Byzantine consensus and its application to blockchains. In *Symposium on Network Computing and Applications (NCA)*, 2018.

25 Danny Dolev, Joseph Y. Halpern, Barbara Simons, and Ray Strong. Dynamic fault-tolerant clock synchronization. *J. ACM*, 42(1):143–185, 1995.

26 Partha Dutta, Rachid Guerraoui, and Leslie Lamport. How fast can eventual synchrony lead to consensus? In *Conference on Dependable Systems and Networks (DSN)*, 2005.

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

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

29 Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony. In *Symposium on Principles of Distributed Computing (PODC)*, 1998.

30 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Symposium on Operating Systems Principles (SOSP)*, 2017.

31 Guy Golan-Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K. Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *Conference on Dependable Systems and Networks (DSN)*, 2019.

32 Andreas Haeberlen and Petr Kuznetsov. The fault detection problem. In *Conference on Principles of Distributed Systems (OPODIS)*, 2009.

33 Idit Keidar and Alexander Shraer. Timeliness, failure-detectors, and consensus performance. In *Symposium on Principles of Distributed Computing (PODC)*, 2006.

34 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

35 Dahlia Malkhi and Michael Reiter. Unreliable intrusion detection in distributed computations. In *Workshop on Computer Security Foundations (CSFW)*, 1997.

36 Zarko Milosevic, Martin Biely, and André Schiper. Bounded delay in Byzantine-tolerant state machine replication. In *Symposium on Reliable Distributed Systems (SRDS)*, 2013.

37 Achour Mostéfaoui and Michel Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *Symposium on Distributed Computing (DISC)*, 1999.

38 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. In *Cryptoeconomics Systems Conference (CES)*, 2020.

39 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear Byzantine SMR. In *Symposium on Distributed Computing (DISC)*, 2020.

40 Barbara Simons, Jennifer Welch, and Nancy Lynch. An overview of clock synchronization. In *Fault-Tolerant Distributed Computing*, 1986.

41 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Symposium on Principles of Distributed Computing (PODC)*, 2019.