Liveness and Latency of Byzantine State-Machine Replication

Manuel Bravo
Informal Systems, Madrid, Spain

Gregory Chockler
University of Surrey, UK

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Abstract

Byzantine state-machine replication (SMR) ensures the consistency of replicated state in the presence of malicious replicas and lies at the heart of the modern blockchain technology. Byzantine SMR protocols often guarantee safety under all circumstances and liveness only under synchrony. However, guaranteeing liveness even under this assumption is nontrivial. So far we have lacked systematic ways of incorporating liveness mechanisms into Byzantine SMR protocols, which often led to subtle bugs. To close this gap, we introduce a modular framework to facilitate the design of provably live and efficient Byzantine SMR protocols. Our framework relies on a *view* abstraction generated by a special *SMR synchronizer* primitive to drive the agreement on command ordering. We present a simple formal specification of an SMR synchronizer and its bounded-space implementation under partial synchrony. We also apply our specification to prove liveness and analyze the latency of three Byzantine SMR protocols via a uniform methodology. In particular, one of these results yields what we believe is the first rigorous liveness proof for the algorithmic core of the seminal PBFT protocol.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Replication, blockchain, partial synchrony, liveness

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.12

Related Version *Extended Version*: `https://arxiv.org/abs/2202.06679` [17]

Funding This work was partially supported by an ERC Starting Grant RACCOON and by a research grant from Nomadic Labs and the Tezos Foundation.

Acknowledgements We thank the following people for comments that helped improve the paper: Lăcrămioara Aştefănoaei, Hagit Attiya, Alysson Besani, Armando Castañeda, Peter Davies, Dan O'Keeffe, Idit Keidar, Giuliano Losa, Alejandro Naser, and Eugen Zălinescu.

1 Introduction

Byzantine state-machine replication (SMR) [52] ensures the consistency of replicated state even when some of the replicas are malicious. It lies at the heart of the modern blockchain technology and is closely related to the classical Byzantine consensus problem. Unfortunately, no deterministic protocol can guarantee both safety and liveness of Byzantine SMR when the network is asynchronous [33]. A common way to circumvent this while maintaining determinism is to guarantee safety under all circumstances and liveness only under synchrony. This is formalized by the *partial synchrony* model [26, 32], which stipulates 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 messages can be lost or delayed, and clocks at different processes can drift apart.


© Manuel Bravo, Gregory Chockler, and Alexey Gotsman;
licensed under Creative Commons License CC-BY 4.0

Historically, researchers have paid more attention to safety of Byzantine SMR protocols than their liveness. For example, while the seminal PBFT protocol came with a detailed safety proof [23, §A], the nontrivial mechanisms ensuring its liveness were only given a brief informal justification [25, §4.5.1], which did not cover their most critical properties. However, ensuring liveness under partial synchrony is far from trivial, as illustrated by the many liveness bugs found in existing protocols [2, 4, 8, 12, 22]. In particular, classical failure detectors and leader oracles [26, 34] are of little help: while they have been widely used under benign failures [38, 39, 47], their implementations under Byzantine failures are either impractical [43] or detect only restricted failure types [30, 40, 46]. As an alternative, a textbook by Cachin et al. [20] proposed a leader oracle-like abstraction that accepts hints from the application to identify potentially faulty processes. However, as we explain in §8 and [17, §F], their specification of the abstraction is impossible to implement, and in fact, the consensus algorithm constructed using it in [20] also suffers from a liveness bug.

Recent work on ensuring liveness has departed from failure detectors and instead revisited the approach of the original DLS paper [32]. This exploits the common structure of Byzantine consensus and SMR protocols under partial synchrony: such protocols usually divide their execution into *views*, each with a designated leader process that coordinates the protocol execution. If the leader is faulty, the processes switch to another view with a different leader. To ensure liveness, an SMR protocol needs to spend sufficient time in views that are entered by all correct processes and where the leader correctly follows the protocol. The challenge of achieving such *view synchronization* is that, before GST, clocks can diverge and messages that could be used to synchronize processes can get lost or delayed; even after GST, Byzantine processes may try to disrupt attempts to bring everybody into the same view. *View synchronizers* [16, 48, 49, 57] encapsulate mechanisms for dealing with this challenge, allowing them to be reused across protocols.

View synchronizers have been mostly explored in the context of (single-shot) Byzantine consensus. In this case a synchronizer can just switch processes through an infinite series of views, so that eventually there is a view with a correct leader that is long enough to reach a decision [16, 49]. However, using such a synchronizer for SMR results in suboptimal solutions. For example, one approach is to use the classical SMR construction where each command is decided using a separate black-box consensus instance [52], implemented using a view synchronizer. However, this would force the processes in every instance to iterate over the same sequence of potentially bad views until the one with a correct leader and sufficiently long duration could be reached. As we discuss in §8, other approaches for implementing SMR based on this type of synchronizers also suffer from drawbacks.

To minimize the overheads of view synchronization, instead of automatically switching processes through views based on a fixed schedule, implementations such as PBFT allow processes to stay in the same view for as long as they are happy with its performance. The processes can then reuse a single view to decide multiple commands, usually with the same leader. To be useful for such SMR protocols, a synchronizer needs to allow the processes to control when they want to switch views via a special `advance` call. We call such a primitive an *SMR synchronizer*, to distinguish it from the less flexible *consensus synchronizer* introduced above. This kind of synchronizers was first introduced in [48, 49], but only used as an intermediate module to implement a consensus synchronizer.

In this paper we show that SMR synchronizers can be *directly* exploited to construct efficient and provably live SMR protocols and develop a general blueprint that enables such constructions. In more detail:

- We propose a formal specification of an SMR synchronizer (§3), which is simpler and more general than prior proposals [48, 49]. It is also strictly stronger than the consensus synchronizer of [16], which can be obtained from the SMR synchronizer at no extra cost. Informally, our specification guarantees that *(a)* the system will move to a new view if enough correct processes call `advance`, and *(b)* all correct processes will enter the new view, provided that for long enough, no correct process that enters this view asks to leave it. These properties enable correct processes to iterate through views in search of a well-behaved leader, and to synchronize in a view they are happy with.
- We give an SMR synchronizer implementation and prove that it satisfies our specification (§3.1). Unlike prior implementations [49], ours tolerates message loss before GST while using only bounded space; in practice, this feature is essential to defend against denial-of-service attacks. We also provide a precise latency analysis of our synchronizer, quantifying how quickly all correct processes enter the next view after enough of them call `advance`.
- We demonstrate the usability of our synchronizer specification by applying it to construct and prove the correctness of several SMR protocols. First, we prove the liveness of a variant of PBFT using an SMR synchronizer (§4-5): to the best of our knowledge, this is the first rigorous proof of liveness for PBFT's algorithmic core. The proof establishes a strong liveness guarantee that implies censorship-resistance: every command submitted by a correct process will be executed. The use of the synchronizer specification in the proof allows us to abstract from view synchronization mechanics and focus on protocol-specific reasoning. This reasoning is done using a reusable methodology based on showing that the use of timers in the SMR protocol and the synchronizer together establish properties similar to those of failure detectors. The methodology also handles the realistic ways in which protocols such as PBFT adapt their timeouts to the unknown message delay δ. We demonstrate the generality of our methodology by also applying it to a version of PBFT with periodic leader changes [28, 55, 56] and a HotStuff-like protocol [57] (§7).
- We exploit the latency bounds for our synchronizer to establish both bad-case and good-case bounds for variants of PBFT implemented on top of it (§6). Our bad-case bound assumes that the protocol starts before GST; it shows that after GST all correct processes synchronize in the same view within a bounded time. This time is proportional to a conservatively chosen constant Δ that bounds post-GST message delays in all executions [41, 50]. Our good-case bound quantifies decision latency when the protocol starts after GST and matches the lower bound of [5].

2 System Model

We consider a system of $n = 3f + 1$ processes. At most f of these can be Byzantine (aka *faulty*), i.e., can behave arbitrarily. The rest of the processes are *correct* and we denote their set by $\mathcal{C}$. We call a set Q of $2f + 1$ processes a *quorum* and write $\mathsf{quorum}(Q)$. We assume standard cryptographic primitives [20, §2.3]: processes can communicate via authenticated point-to-point links, sign messages using digital signatures, and use a collision-resistant hash function $\mathsf{hash}()$. We denote by $\langle m \rangle_i$ a message m signed by process p_i.

We consider a *partial synchrony* model [26, 32]: for each execution of the protocol, there exist a time GST and a duration δ such that after GST message delays between correct processes are bounded by δ; before GST messages can get arbitrarily delayed or lost. As in [26], we assume that the values of GST and δ are unknown to the protocol. This 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 processes have hardware clocks that can drift unboundedly from real time before GST, but do not drift thereafter.

1. **Monotonicity.** A process enters increasing views:
   $\forall i, v, v'.\, E_i(v)\downarrow \wedge\, E_i(v')\downarrow \implies (v < v' \iff E_i(v) < E_i(v'))$
2. **Validity.** A process only enters $v+1$ if some correct process has attempted to advance from v:
   $\forall i, v.\, E_i(v+1)\downarrow \implies A_{\text{first}}(v)\downarrow \wedge\, A_{\text{first}}(v) < E_i(v+1)$
3. **Bounded Entry.** For some $\mathcal{V}$ and d, if a process enters a view $v \geq \mathcal{V}$ and no process attempts to advance to a higher view within time d, then all correct processes will enter v within d:
   $\exists \mathcal{V}, d.\forall v \geq \mathcal{V}.\, E_{\text{first}}(v)\downarrow \wedge \neg(A_{\text{first}}(v) < E_{\text{first}}(v)+d) \implies (\forall p_i \in \mathcal{C}.\, E_i(v)\downarrow) \wedge E_{\text{last}}(v) \leq E_{\text{first}}(v)+d$
4. **Startup.** Some correct process will enter view 1 if $f+1$ processes call `advance`:
   $(\exists P \subseteq \mathcal{C}.\, |P| = f+1 \wedge (\forall p_i \in P.\, A_i(0)\downarrow)) \implies E_{\text{first}}(1)\downarrow$
5. **Progress.** If a correct process enters a view v and, for some set P of $f+1$ correct processes, any process in P that enters v eventually calls `advance`, then some correct process will enter $v+1$:
   $\forall v.\, E_{\text{first}}(v)\downarrow \wedge\, (\exists P \subseteq \mathcal{C}.\, |P| = f+1 \wedge (\forall p_i \in P.\, E_i(v)\downarrow \implies A_i(v)\downarrow)) \implies E_{\text{first}}(v+1)\downarrow$

Figure 1 SMR synchronizer specification.

3 SMR Synchronizer Specification and Implementation

We consider a synchronizer interface defined in [48, 49], which here we call an *SMR synchronizer*. Let $\mathsf{View} = \{1, 2, \ldots\}$ be the set of *views*, ranged over by v; we use 0 to denote an invalid initial view. The synchronizer produces notifications `new_view`(v) at a process, telling it to *enter* view v. To trigger these, the synchronizer allows a process to call a function `advance`(), which signals that the process wishes to *advance* to a higher view. We assume that a correct process does not call `advance` twice without an intervening `new_view` notification.

Our first contribution is the SMR synchronizer specification in Figure 1, which is simpler and more general than prior proposals [48, 49] (see §8 for a discussion). The specification relies on the following notation. Given a view v 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 denote by $A_i(v)$ the time when a correct process p_i calls `advance` while in v, and let $A_{\text{first}}(v)$ and $A_{\text{last}}(v)$ denote respectively the earliest and the latest time when this happens. Given a partial function f, we write $f(x)\downarrow$ if $f(x)$ is defined, and $f(x)\uparrow$ if $f(x)$ is undefined.

The Monotonicity property in Figure 1 ensures that views can only increase at a given process. Validity ensures that a process may only enter a view $v+1$ if some correct process has called `advance` in v. This prevents faulty processes from disrupting the system by forcing view changes. As a corollary of Validity we can prove that, if a view v' is entered by some correct process, then so are all the views v preceding v'.

▶ **Proposition 1.** $\forall v, v'.\, 0 < v < v' \wedge E_{\text{first}}(v')\downarrow \implies E_{\text{first}}(v)\downarrow \wedge E_{\text{first}}(v) < E_{\text{first}}(v')$.

Proof. Fix $v' \geq 2$ and assume that a correct process enters v', so that $E_{\text{first}}(v')\downarrow$. We prove by induction on k that $\forall k = 0..(v'-1).\, E_{\text{first}}(v'-k)\downarrow \wedge E_{\text{first}}(v'-k) \leq E_{\text{first}}(v')$. The base case of $k = 0$ is trivial. For the inductive step, assume that the required holds for some k. Then by Validity there exists a time $t < E_{\text{first}}(v'-k)$ at which some correct process p_j attempts to advance from $v'-k-1$. But then p_j's view at t is $v'-k-1$. Hence, p_j enters $v'-k-1$ before t, so that $E_{\text{first}}(v'-k-1) < t < E_{\text{first}}(v'-k) \leq E_{\text{first}}(v')$, as required. ◀

Bounded Entry ensures that, if some process enters view v, then all correct processes will do so within at most d time units of each other ($d = 2\delta$ for our implementation). This only holds if within d no process attempts to advance to a higher view, as this may make some processes skip v and enter a higher view directly. Bounded Entry also holds only starting from some view $\mathcal{V}$, since we may not be able to guarantee it for views entered before GST.

```
1 when the process starts or timer expires
2     advance();

3 upon new_view(v)
4     stop_timer(timer);
5     start_timer(timer, τ);
```

Figure 2 A simple client of the SMR synchronizer.

Startup ensures that some correct process enters view 1 if $f+1$ processes call `advance`. Given a view v entered by a correct process, Progress determines conditions under which some correct process will enter the next view $v+1$. This will happen if for some set P of $f+1$ correct processes, any process in P entering v eventually calls `advance`. Note that even a single `advance` call at a correct process *may* lead to a view switch (reflecting the fact that in implementations faulty processes may help this correct process). Startup and Progress ensure that the synchronizer *must* switch if at least $f+1$ correct processes ask for this. We now illustrate a typical pattern of their use, which we later apply to PBFT (§5). To this end, we consider a simple client in Figure 2, where in each view a process sets a timer for a fixed duration τ and calls `advance` when the timer expires. Using Startup and Progress we prove that this client keeps switching views forever as follows.

▶ **Proposition 2.** *In any execution of the client in Figure 2:* $\forall v.\, \exists v'.\, v' > v \wedge E_{\text{first}}(v')\downarrow$.

Proof. Since all correct processes initially call `advance`, by Startup some correct process eventually enters view 1. Assume now that the proposition is false, so that there is a maximal view v entered by any correct process. Let P be any set of $f+1$ correct processes and consider an arbitrary process $p_i \in P$ that enters v. When this happens, p_i sets the timer for the duration τ. The process then either calls `advance` when timer expires, or enters a new view v' before this. In the latter case $v' > v$ by Monotonicity, which is impossible. Hence, p_i eventually calls `advance` while in v. Since p_i was chosen arbitrarily, $\forall p_i \in P.\, E_i(v)\downarrow \implies A_i(v)\downarrow$. Then by Progress we get $E_{\text{first}}(v+1)\downarrow$: a contradiction. ◀

Similarly to Figure 2, we can use an SMR synchronizer satisfying the properties in Figure 1 to implement a *consensus synchronizer* [16, 49] without extra overhead. This lacks an `advance` call and provides only the `new_view` notification, which it keeps invoking at increasing intervals so that eventually the there is a view long enough for the consensus protocol running on top to decide. We obtain a consensus synchronizer if in Figure 2 we propagate the `new_view` notification to the consensus protocol and set the timer to an unboundedly increasing function of views instead of a constant τ. In [17, §A] we show that the resulting consensus synchronizer satisfies the specification proposed in [16].

3.1 A Bounded-Space SMR Synchronizer

We now present a bounded-space algorithm that implements the specification in Figure 1 under partial synchrony for $d = 2\delta$. Our implementation reuses algorithmic techniques from a consensus synchronizer of Bravo et al. [16]. However, it supports a more general abstraction, and thus requires a more intricate correctness proof and latency analysis (§3.2).

When a process calls `advance` (line 1), the synchronizer does not immediately move to the next view v', but disseminates a `WISH`(v') message announcing its intention. A process enters a new view once it accumulates a sufficient number of `WISH` messages supporting this. A naive synchronizer design could follow Bracha broadcast [15]: enter a view v' upon receiving

```
1  function advance()
2      send WISH(max(view + 1, view⁺)) to all;
3      advanced ← TRUE;

4  periodically                          ▷ every ρ time units
5      if advanced then
6          send WISH(max(view + 1, view⁺)) to all;
7      else if view⁺ > 0 then
8          send WISH(view⁺) to all;

9  when received WISH(v) from p_j
10     prev_v, prev_v⁺ ← view, view⁺;
11     if v > max_views[j] then max_views[j] ← v;
12     view  ← max{v | ∃k. max_views[k] = v ∧
                   |{j | max_views[j] ≥ v}| ≥ 2f + 1};
13     view⁺ ← max{v | ∃k. max_views[k] = v ∧
                   |{j | max_views[j] ≥ v}| ≥ f + 1};
14     if view⁺ = view ∧ view > prev_v then
15         trigger new_view(view);
16         advanced ← FALSE;
17     if view⁺ > prev_v⁺ then
18         send WISH(view⁺) to all
```

Figure 3 A bounded-space SMR synchronizer. All counters are initially 0.

$2f + 1$ WISH(v') messages, and echo WISH(v') upon receiving $f + 1$ copies thereof; the latter is needed to combat equivocation by Byzantine processes. However, in this case the process would have to track all newly proposed views for which $< 2f + 1$ WISHes have been received. Since messages sent before GST can be lost or delayed, this would require unbounded space. To reduce the space complexity, in our algorithm a process only remembers the highest view received from each process, kept in an array max_views (line 11). Variables view and view$^+$ respectively hold the $(2f + 1)$st highest and the $(f + 1)$st highest views in max_views (lines 12-13). These variables never decrease and always satisfy view $\leq$ view$^+$.

The process enters the view stored in view when this variable increases (line 15). A process thus enters v only if it receives $2f + 1$ WISHes for views $\geq v$, and a process may be forced to switch views even if it did not call advance; the latter helps lagging processes to catch up. The variable view$^+$ increases when the process receives $f + 1$ WISHes for views $\geq$ view$^+$, and thus some correct process wishes to enter a new view $\geq$ view$^+$. In this case we echo view$^+$ (line 18), to help other processes switch views and satisfy Bounded Entry and Progress.

The guard view$^+$ = view in line 14 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 process calls advance, it sends a WISH for the maximum of view + 1 and view$^+$ (line 2). Thus, if view = 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 next view (view + 1). Otherwise, view $<$ view$^+$, and the process sends a WISH for the higher view view$^+$. Finally, to deal with message loss before GST, a process retransmits the highest WISH it sent every ρ time units, according to its local clock (line 4). Depending on whether the process has called advance in the current view (tracked by advanced), the WISH is computed as in lines 18 or 2.

Our SMR synchronizer requires only $O(n)$ variables for storing views. Proposition 1 also ensures that views entered by correct processes do not skip values, which limits the power of the adversary to exhaust their allocated space (similarly to [11]).

3.2 SMR Synchronizer Correctness and Latency Bounds

The following theorem (proved in [17, §B]) states the correctness of our synchronizer as well as and its performance properties. In §6 we apply the latter to bound the latency of Byzantine SMR protocols. Given a view v that was entered by a correct process p_i, we let $T_i(v)$ denote the time at which p_i either attempts to advance from v or enters a view $> v$; we let $T_{\text{last}}(v)$ denote the latest time when a correct process does so. We assume that every correct process eventually attempts to advance from view 0 unless it enters a view > 0, i.e., $\forall p_i \in \mathcal{C}. T_i(0)\downarrow$.

▶ **Theorem 3.** *Consider an execution with an eventual message delay δ. The algorithm in Figure 3 satisfies the properties in Figure 1 for $d = 2\delta$ and $\mathcal{V} = \max\{v \mid (E_{\text{first}}(v)\downarrow \wedge E_{\text{first}}(v) < \mathsf{GST} + \rho) \vee v = 0\} + 1$ if $A_{\text{first}}(0) < \mathsf{GST}$, and $\mathcal{V} = 1$, otherwise. Furthermore:*

A. $\forall v.\, E_{\text{first}}(v)\downarrow \wedge\ A_{\text{first}}(0) < \mathsf{GST} \implies E_{\text{last}}(v) \leq \max(E_{\text{first}}(v), \mathsf{GST} + \rho) + 2\delta.$

B. $\forall v.\, E_{\text{first}}(v+1)\downarrow \implies E_{\text{last}}(v+1) \leq \begin{cases} \max(T_{\text{last}}(v), \mathsf{GST} + \rho) + \delta, & \text{if } A_{\text{first}}(0) < \mathsf{GST}; \\ T_{\text{last}}(v) + \delta, & \text{otherwise.} \end{cases}$

The theorem gives a witness for $\mathcal{V}$ in Bounded Entry: it is the next view after the highest one entered by a correct process at or before $\mathsf{GST}+\rho$ (or 1 if no view was entered). Property A bounds the latest time any correct process can enter a view that has been previously entered by a correct process. It is similar to Bounded Entry, but also handles views $< \mathcal{V}$. Property B refines Progress: while the latter guarantees that the synchronizer will enter $v+1$ if enough processes ask for this, the former bounds the time by which this will happen.

4 PBFT Using an SMR Synchronizer

We now demonstrate how an SMR synchronizer can be used to implement Byzantine SMR. More formally, we implement *Byzantine atomic broadcast* [20], from which SMR can be implemented in the standard way [52]. This allows processes to broadcast *values*, and we assume an application-specific predicate to indicate whether a value is valid [21] (e.g., a block in a blockchain is invalid if it lacks correct signatures). We assume that all values broadcast by correct processes in a single execution are valid and unique. Then Byzantine atomic broadcast is defined by the following properties:

- **Integrity.** Every process delivers a value at most once.
- **External Validity.** A correct process delivers only values satisfying $\mathsf{valid}()$.
- **Ordering.** If a correct process p delivers x_1 before x_2, then another correct process q cannot deliver x_2 before x_1.
- **Liveness.** If a correct process broadcasts or delivers x, then eventually all correct processes will deliver x. (Note that this implies *censorship-resistance*: the service cannot selectively omit values submitted by correct processes.)

The PBFT-light protocol. We implement Byzantine atomic broadcast in a *PBFT-light* protocol (Figures 4-6), which faithfully captures the algorithmic core of the seminal Practical Byzantine Fault Tolerance protocol (PBFT) [24]. Whereas PBFT integrated view synchronization functionality with the core SMR protocol, PBFT-light delegates this to an SMR synchronizer, and in §5 we rigorously prove its liveness when using any synchronizer satisfying our specification. When using the synchronizer in Figure 3, the protocol also incurs only bounded space overhead (see [17, §C.4] for details).

We base PBFT-light on the PBFT protocol with signatures and, for simplicity, omit the mechanisms for managing checkpoints and watermarks; these can be easily added without affecting liveness. The protocol works in a succession of views produced by the synchronizer. A process stores its current view in $\mathsf{curr\_view}$. Each view v has a fixed *leader* $\mathsf{leader}(v) = p_{((v-1) \bmod n)+1}$ that is responsible for totally ordering values submitted for broadcast; the other processes are *followers*, which vote on proposals made by the leader. Processes store the sequence of (unique) values proposed by the leader in a log array; at the leader, a next counter points to the first free slot in the array. Processes monitor the leader's behavior and ask the synchronizer to advance to another view if they suspect that the leader is faulty. A status variable records whether the process is operating as normal in the current view (NORMAL) or is changing the view.

```
1  function start()
2      if curr_view = 0 then advance();
3  when a timer expires
4      stop_all_timers();
5      advance();
6      status ← ADVANCED;
7      dur_delivery ← dur_delivery + τ;
8      dur_recovery ← dur_recovery + τ;
9  function broadcast(x)
10     pre: valid(x);
11     send ⟨BROADCAST(x)⟩_i to all
         periodically until x is delivered
12 when received BROADCAST(x)
13     pre: valid(x) ∧ status = NORMAL ∧
            (timer_delivery[x] not active) ∧
            (∀k. k ≤ last_delivered ⟹
             commit_log[k] ≠ x);
14     start_timer(timer_delivery[x],
         dur_delivery);
15     send ⟨FORWARD(x)⟩_i to
         leader(curr_view);
16 when received FORWARD(x)
17     pre: valid(x) ∧ status = NORMAL ∧
            p_i = leader(curr_view) ∧
            ∀k. log[k] ≠ x;
18     send ⟨PREPREPARE(curr_view,
         next, x)⟩_i to all;
19     next ← next + 1;
20 when received ⟨PREPREPARE(v, k, x)⟩_j
21     pre: p_j = leader(v) ∧ curr_view = v ∧
            status = NORMAL ∧ phase[k] = START ∧
            valid(x) ∧ (∀k'. log[k'] ≠ x)
22     (log, phase)[k] ← (x, PREPREPARED);
23     send ⟨PREPARE(v, k, hash(x))⟩_i to all;
24 when received {⟨PREPARE(v, k, h)⟩_j | p_j ∈ Q} = C
     for a quorum Q
25     pre: curr_view = v ∧ phase[k] = PREPREPARED ∧
            status = NORMAL ∧ hash(log[k]) = h;
26     (prep_log, prep_view, cert, phase)[k] ←
         (log[k], curr_view, C, PREPARED);
27     send ⟨COMMIT(v, k, h)⟩_i to all;
28 when received {⟨COMMIT(v, k, h)⟩_j | p_j ∈ Q} = C
     for a quorum Q
29     pre: curr_view = v ∧ phase[k] = PREPARED ∧
            status = NORMAL ∧ hash(prep_log[k]) = h;
30     (commit_log, phase)[k] ← (log[k], COMMITTED);
31     broadcast ⟨DECISION(commit_log[k], k, C);
32 when commit_log[last_delivered + 1] ≠ ⊥
33     last_delivered ← last_delivered + 1;
34     if commit_log[last_delivered] ≠ nop then
35         deliver(commit_log[last_delivered])
36     stop_timer(
         timer_delivery[commit_log[last_delivered]]);
37     if last_delivered = init_log_length ∧
         status = NORMAL then
38         stop_timer(timer_recovery);
39 when received DECISION(x, k, C)
40     pre: commit_log[k] ≠ ⊥ ∧
            ∃v. committed(C, v, k, hash(x));
41     commit_log[k] ← x;
```

Figure 4 Normal operation of PBFT-light at a process p_i.

Normal protocol operation. A process broadcasts a valid value x using a `broadcast` function (line 9). This keeps sending the value to all processes in a `BROADCAST` message until the process delivers the value, to tolerate message loss before GST. When a process receives a `BROADCAST` message with a new value (line 12), it forwards the value to the leader in a `FORWARD` message. This ensures that the value reaches the leader even when broadcast by a faulty process, which may withhold the `BROADCAST` message from the leader. (We explain the timer set in line 14 later.) When the leader receives a new value x in a `FORWARD` message (line 16), it sends a `PREPREPARE` message to all processes (including itself) that includes x and its position in the log, generated from the next counter. Processes vote on the leader's proposal in two phases. Each process keeps track of the status of values going through the vote in an array phase, whose entries initially store START.

When a process receives a proposal x for a position k from the leader of its view v (line 20), it first checks that phase$[k]$ = START, so that it has not yet accepted a proposal for the position k in the current view. It also checks that the value is valid and distinct from all values it knows about. The process then stores x in log$[k]$ and advances phase$[k]$ to PREPREPARED. Since a faulty leader may send different proposals for the same position to different processes, the process next communicates with others to check that they received the same proposal.

To this end, it disseminates a PREPARE message with the position and the hash of the value x it received. The process handles x further once it gathers a set C of PREPARE messages from a quorum matching the value (line 24), which we call a *prepared certificate* and check using the prepared predicate in Figure 6. In this case the process stores the value in prep_log$[k]$, the certificate in cert$[k]$, and the view in which it was formed in prep_view$[k]$. At this point we say that the process *prepared* the proposal, as recorded by setting its phase to PREPARED. It is easy to show that processes cannot prepare different values at the same position and view, since each correct process can send only one corresponding PREPARE message.

Having prepared a value, the process disseminates a COMMIT message with its hash. Once the process gathers a quorum of matching COMMIT messages (line 28), it stores the value in a commit_log array and advances its phase to COMMITTED: the value is now *committed*. The protocol ensures that correct processes cannot commit different values at the same position, even in different views. We call a quorum of matching COMMIT messages a *commit certificate* and check it using the committed predicate in Figure 6. A process delivers committed values in the commit_log order, with last_delivered tracking the position last delivered position.

To satisfy the Liveness property of atomic broadcast, similarly to [12], PBFT-light allows a process to find out about committed values from other processes directly. When a process commits a value (line 28), it disseminates a DECISION message with the value, its position k in the log and the commit certificate (line 31). A process receiving a DECISION with a valid certificate saves the value in commit_log$[k]$, which allows it to be delivered (line 32). The DECISION messages are disseminated via reliable broadcast ensuring that, if one correct process delivers the value, then so do all others. To implement this, each process could periodically resend the DECISION messages it has (omitted from the pseudocode). A more practical implementation would only resend information that other processes are missing. As proved in [32], such periodic resends are unavoidable in the presence of message loss.

View initialization. When the synchronizer tells a process to move to a new view v (line 42), the process sets curr_view to v, which ensures that it will no longer accept messages from prior views. It also sets status to INITIALIZING, which means that the process is not yet ready to order values in the new view. It then sends a NEW_LEADER message to the leader of v with the information about the values it has prepared so far and their certificates[1].

The new leader waits until it receives a quorum of well-formed NEW_LEADER messages, as checked by the predicate ValidNewLeader (line 48). Based on these, the leader computes the initial log of the new view, stored in log'. Similarly to Paxos [45], for each index k the leader puts at the kth position in log' the value prepared in the highest view (line 50). The resulting array may contain empty or duplicate entries. To resolve this, the leader writes nop into empty entries and those entries for which there is a duplicate prepared in a higher view (line 53). The latter is safe because one can show that no value could have been committed in such entries in prior views. Finally, the leader sends a NEW_STATE message to all processes, containing the initial log and the NEW_LEADER messages from which it was computed (line 56).

A process receiving a NEW_STATE first checks its correctness by redoing the leader's computation (ValidNewState, line 57). If the check passes, the process overwrites its log with the new one and sets status to NORMAL. It also sends PREPARE messages for all log entries, to commit them in the new view. A more practical implementation would include a checkpointing mechanism, so that a process restarts committing previous log entries only from the last stable checkpoint [24]; this mechanism can be easily added to PBFT-light.

[1] In PBFT this information is sent in VIEW-CHANGE messages, which also play a role similar to WISH messages in our synchronizer (Figure 3). In PBFT-light we opted to eschew VIEW-CHANGE messages to maintain a clear separation between view synchronization internals and the SMR protocol.

```
42 upon new_view(v)
43     stop_all_timers();
44     curr_view ← v;
45     status ← INITIALIZING;
46     send ⟨NEW_LEADER(curr_view, prep_view,
         prep_log, cert)⟩_i to leader(curr_view);
47     start_timer(timer_recovery, dur_recovery);

48 when received {⟨NEW_LEADER(v, prep_view_j,
     prep_log_j, cert_j)⟩_j | p_j ∈ Q} = M
     for a quorum Q
49     pre: p_i = leader(v) ∧ curr_view = v ∧
            status = INITIALIZING ∧
            ∀m ∈ M. ValidNewLeader(m);
50     forall k do
51         if ∃p_j' ∈ Q. prep_view_j'[k] ≠ 0 ∧
             ∀p_j ∈ Q. prep_view_j[k] ≤ prep_view_j'[k]
             then log'[k] ← prep_log_j'[k];
52     next ← max{k | log'[k] ≠ ⊥};
53     forall k = 1..(next − 1) do
54         if log'[k] = ⊥ ∨ ∃k'. k' ≠ k ∧
             log'[k'] = log'[k] ∧ ∃p_j' ∈ Q. ∀p_j ∈ Q.
             prep_view_j'[k'] > prep_view_j[k] then
55             log'[k] ← nop
56     send ⟨NEW_STATE(v, log', M)⟩_i to all;

57 when received ⟨NEW_STATE(v, log', M)⟩_j = m
58     pre: status = INITIALIZING ∧
            curr_view = v ∧ ValidNewState(m);
59     log ← log';
60     forall {k | log[k] ≠ ⊥} do
61         phase[k] ← PREPREPARED;
62         send ⟨PREPARE(v, k, hash(log[k]))⟩_i
             to all;
63     status ← NORMAL;
64     init_log_length ← max{k | log[k] ≠ ⊥};
65     if init_log_length ≤ last_delivered then
66         stop_timer(timer_recovery);
```

Figure 5 View-initialization protocol of PBFT-light at a process p_i.

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

$$\mathsf{committed}(C, v, k, h) \iff \exists Q.\, \mathsf{quorum}(Q) \wedge C = \{\langle \texttt{COMMIT}(v, k, h)\rangle_j \mid p_j \in Q\}$$

$$\mathsf{ValidNewLeader}(\langle \texttt{NEW\_LEADER}(v, prep\_view, prep\_log, cert)\rangle\_) \iff$$
$$\forall k.\, (prep\_view[k] > 0 \implies prep\_view[k] < v \wedge \mathsf{prepared}(cert[k], prep\_view[k], k, prep\_log[k]))$$

$$\mathsf{ValidNewState}(\langle \texttt{NEW\_STATE}(v, log', M)\rangle_i) \iff p_i = \mathsf{leader}(v) \wedge \exists Q, prep\_view, prep\_log, cert.$$
$$\mathsf{quorum}(Q) \wedge M = \{\langle \texttt{NEW\_LEADER}(v, prep\_view_j, log_j, cert_j)\rangle_j \mid p_j \in Q\} \wedge$$
$$(\forall m \in C.\, \mathsf{ValidNewLeader}(m)) \wedge (log' \text{ is computed from } M \text{ as per lines 50-55})$$

Figure 6 Auxiliary predicates for PBFT-light.

Triggering view changes. We now describe when a process calls `advance`, which is key to ensure liveness (§5). This happens either on start-up (line 2) or when the process suspects that the current leader is faulty. To this end, the process monitors the leader's behavior using timers; if one of these expires, the process calls `advance` and sets status to ADVANCED (line 3). First, the process checks that each value it receives is delivered promptly: e.g., to guard against a faulty leader censoring certain values. For a value x this is done using timer_delivery$[x]$, set for a duration dur_delivery when the process receives $\texttt{BROADCAST}(x)$ (lines 14). The timer is stopped when the process delivers x (line 36). A process also checks that the leader initializes a view quickly enough: e.g., to guard against the leader crashing during the initialization. Thus, when a process enters a view it starts timer_recovery for a duration dur_recovery (line 47). The process stops the timer when it delivers all values in the initial log (lines 38 and 66). The above checks may make a process suspect a correct leader if the timeouts are initially set too small with respect to the message delay δ, unknown to the process. To deal with this, a process increases dur_delivery and dur_recovery each time a timer expires, which signals that the current view is not operating normally (lines 7-8).

5 Proving the Liveness of PBFT

Assume that PBFT-light is used with a synchronizer satisfying the specification in Figure 1; to simplify the following latency analysis we let $d = 2\delta$, as for the synchronizer in Figure 3. We now prove that the protocol satisfies the Liveness property of Byzantine atomic broadcast; we defer the proof of the other properties to [17, §C.1]. To the best of our knowledge, this is the first rigorous proof of liveness for the algorithmic core of PBFT: as we elaborate in §8, the liveness mechanisms of PBFT came only with a brief informal justification, which did not cover their most critical properties [25, §4.5.1]. Our proof is simplified by the use of the synchronizer specification, which allows us to abstract from view synchronization mechanics.

We prove the liveness of PBFT-light by showing that the protocol establishes properties reminiscent of those of failure detectors [26]. First, similarly to their completeness property, we prove that every correct process eventually attempts to advance from a *bad* view in which no progress is possible (e.g., because the leader is faulty).

▶ **Lemma 4.** *Assume that a correct process p_i receives* `BROADCAST`(x) *for a valid value x while in a view v. If p_i never delivers x and never enters a view higher than v, then it eventually calls* `advance` *in v.*

The lemma holds because in PBFT-light each process monitors the leader's behavior using timers, and we defer its easy proof to [17, §C.2]. Our next lemma is similar to the eventual accuracy property of failure detectors. It stipulates that if the timeout values are high enough, then eventually any correct process that enters a *good* view (with a correct leader) will never attempt to advance from it. Let $\mathsf{dur\_recovery}_i(v)$ and $\mathsf{dur\_delivery}_i(v)$ denote respectively the value of dur_recovery and dur_delivery at a correct process p_i while in view v.

▶ **Lemma 5.** *Consider a view $v \geq \mathcal{V}$ such that $E_{\text{first}}(v) \geq \mathsf{GST}$ and $\mathsf{leader}(v)$ is correct. If $\mathsf{dur\_recovery}_i(v) > 6\delta$ and $\mathsf{dur\_delivery}_i(v) > 4\delta$ at each correct process p_i that enters v, then no correct process calls* `advance` *in v.*

Before proving the lemma, we informally explain the rationale for the bounds on timeouts in it, using the example of dur_recovery. The timer timer_recovery is started at a process p_i when this process enters a view v (line 47), and is stopped when the process delivers all values inherited from previous views (lines 38 or 66). The two events are separated by 4 communication steps of PBFT-light, exchanging messages of the types `NEW_LEADER`, `NEW_STATE`, `PREPARE` and `COMMIT` (Figure 7). However, 4δ would be too small a value for dur_recovery. This is because the leader of v sends its `NEW_STATE` message only after receiving a quorum of `NEW_LEADER` messages, and different processes may enter v and send their `NEW_LEADER` messages at different times (e.g., p_i and p_j in Figure 7). Hence, dur_recovery must additionally accommodate the maximum discrepancy in the entry times, which is $d = 2\delta$ by the Bounded Entry property. Then to ensure that p_i stops the timer before it expires, we require $\mathsf{dur\_recovery}_i(v) > 6\delta$. As the above reasoning illustrates, Lemma 5 is more subtle than Lemma 4: while the latter is ensured just by the checks in the SMR protocol, the former relies on the Bounded Entry property of the synchronizer.

Another subtlety about Lemma 5 is that the δ used in its premise is a priori unknown. Hence, to apply the lemma in the liveness proof of PBFT-light, we have to argue that, if correct processes keep changing views due to lack of progress, then all of them will eventually increase their timeouts high enough to satisfy the bounds in Lemma 5. This is nontrivial due to the fact that, as in the original PBFT [23, §2.3.5], in our protocol the processes update their timeouts independently, and may thus disagree on their durations. For example, the

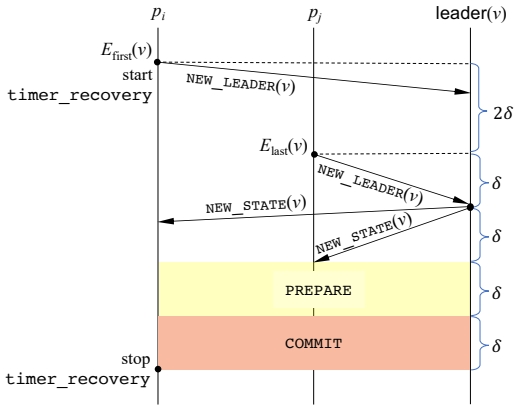

Figure 7 An illustration of the proof of Lemma 5.

first correct process p_i to detect a problem with the current view v will increase its timeouts and call `advance` (line 3). The synchronizer may then trigger `new_view` notifications at other correct processes before they detect the problem as well, so that their timeouts will stay unchanged (line 42). One may think that this allows executions in which only some correct processes keep increasing their timeouts until they are high enough, whereas others are forever stuck with timeouts that are too low, invalidating the premise of Lemma 5. The following lemma rules out such scenarios and also trivially implies Lemma 5. It establishes that, in a sufficiently high view v with a correct leader, if the timeouts at a correct process p_i that enters v are high enough, then this process cannot be the first one to initiate a view change. Hence, for the protocol to enter another view, some other process with lower timeouts must call `advance` and thus increase their durations (line 3).

▶ **Lemma 6.** *Let $v \geq \mathcal{V}$ be such that $E_{\text{first}}(v) \geq \mathsf{GST}$ and $\mathsf{leader}(v)$ is correct, and consider a correct process p_i that enters v. If $\mathsf{dur\_recovery}_i(v) > 6\delta$ and $\mathsf{dur\_delivery}_i(v) > 4\delta$ then p_i is not the first correct process to call* `advance` *in v.*

Proof. Since $E_{\text{first}}(v) \geq \mathsf{GST}$, messages sent by correct processes after $E_{\text{first}}(v)$ get delivered to all correct processes within δ and process clocks track real time. By contradiction, assume that p_i is the first correct process to call `advance` in v. This happens because a timer expires at p_i. Here we only consider the case when it is $\mathsf{timer\_recovery}$, and handle $\mathsf{timer\_delivery}$ in [17, §C.2]. A process starts $\mathsf{timer\_recovery}$ when it enters the view v (line 47), and hence, at $E_{\text{first}}(v)$ at the earliest (Figure 7). Because p_i is the first correct process to call `advance` in v and $\mathsf{dur\_recovery}_i(v) > 6\delta$, no correct process calls `advance` in v until after $E_{\text{first}}(v) + 6\delta$. Then by Bounded Entry all correct processes enter v by $E_{\text{first}}(v) + 2\delta$. Also, by Validity no correct process can enter $v+1$ until after $E_{\text{first}}(v) + 6\delta$, and by Proposition 1 the same holds for any view $> v$. Thus, all correct processes stay in v at least until $E_{\text{first}}(v) + 6\delta$.

When a correct process enters v, it sends a `NEW_LEADER` message to the leader of v, which happens by $E_{\text{first}}(v) + 2\delta$. When the leader receives such messages from a quorum of processes, it broadcasts a `NEW_STATE` message. Thus, by $E_{\text{first}}(v) + 4\delta$ all correct processes receive this message and set $\mathsf{status} = \text{NORMAL}$. If at that point $\mathsf{init\_log\_length} \leq \mathsf{last\_delivered}$ at p_i, then the process stops $\mathsf{timer\_recovery}$ (line 66), which contradicts our assumption. Hence, $\mathsf{init\_log\_length} > \mathsf{last\_delivered}$. When a correct process receives `NEW_STATE`, it sends `PREPARE` messages for all positions $\leq \mathsf{init\_log\_length}$ (line 62). It then takes the correct processes at most 2δ to exchange the sequence of `PREPARE` and `COMMIT` messages that commits the values at all positions $\leq \mathsf{init\_log\_length}$. Thus, by $E_{\text{first}}(v) + 6\delta$ the process p_i commits and delivers all these positions, stopping $\mathsf{timer\_recovery}$ (line 38): a contradiction. ◀

▶ **Theorem 7.** *PBFT-light satisfies the Liveness property of Byzantine atomic broadcast.*

Proof. Consider a valid value x broadcast by a correct process. We first prove that x is eventually delivered by some correct process. Assume the contrary. We show:

▷ Claim 1. Every view is entered by some correct process.

Proof. Since all correct processes call `start` (line 1), by Startup a correct process eventually enters some view. We now show that correct processes keep entering new views forever (analogously to the proof of Proposition 2 in §3). Assume that this is false, so that there exists a maximal view v entered by any correct process. Let P be any set of $f+1$ correct processes and consider an arbitrary process $p_i \in P$ that enters v. The process that broadcast x is correct, and thus keeps broadcasting x until the value is delivered (line 11). Since x is never delivered, p_i is guaranteed to receive x while in v. Then by Lemma 4, p_i eventually calls `advance` while in v. Since p_i was picked arbitrarily, we have $\forall p_i \in P.\, E_i(v)\downarrow \implies A_i(v)\downarrow$. Then by Progress we get $E_{\text{first}}(v+1)\downarrow$, which yields a contradiction. Thus, correct processes keep entering views forever. The claim then follows from Proposition 1. ◁

Let view v_1 be the first view such that $v_1 \geq \mathcal{V}$ and $E_{\text{first}}(v_1) \geq \mathsf{GST}$; such a view exists by Claim 1. The next claim is needed to show that all correct processes will increase their timeouts high enough to satisfy the bounds in Lemma 5.

▷ Claim 2. Every correct process calls the timer expiration handler (line 3) infinitely often.

Proof. Assume the contrary and let C_{fin} and C_{inf} be the sets of correct processes that call the timer expiration handler finitely and infinitely often, respectively. Then $C_{\text{fin}} \neq \emptyset$, and by Claim 1 and Validity, $C_{\text{inf}} \neq \emptyset$. The values of $\mathsf{dur\_delivery}$ and $\mathsf{dur\_recovery}$ increase unboundedly at processes from C_{inf}, and do not change after some view v_2 at processes from C_{fin}. By Claim 1 and since leaders rotate round-robin, there is a view $v_3 \geq \max\{v_2, v_1\}$ with a correct leader such that any process $p_i \in C_{\text{inf}}$ that enters v_3 has $\mathsf{dur\_delivery}_i(v_3) > 4\delta$ and $\mathsf{dur\_recovery}_i(v_3) > 6\delta$. By Claim 1 and Validity, at least one correct process calls `advance` in v_3; let p_l be the first process to do so. Since $v_3 \geq v_2$, p_l cannot be in C_{fin} because none of these processes increase their timers in v_3. Then $p_l \in C_{\text{inf}}$, contradicting Lemma 6. ◁

By Claims 1 and 2, there exists a view $v_4 \geq v_1$ with a correct leader such that some correct process enters v_4, and for any correct process p_i that enters v_4 we have $\mathsf{dur\_delivery}_i(v_4) > 4\delta$ and $\mathsf{dur\_recovery}_i(v_4) > 6\delta$. By Lemma 5, no correct process calls `advance` in v_4. Then, by Validity, no correct process enters v_4+1, which contradicts Claim 1. This contradiction shows that x must be delivered by a correct process. Then, since the protocol reliably broadcasts committed values (line 31), all correct processes will also eventually deliver x. ◀

6 Latency Bounds for PBFT

Assume that PBFT-light is used with our SMR synchronizer in Figure 3. We now quantify its latency using the bounds for the synchronizer in Theorem 3, yielding the first detailed latency analysis for a PBFT-like protocol. Due to space constraints we defer proofs to [17, §C.3]. To state our bounds, we assume the existence of a known upper bound Δ on the maximum value of δ in any execution [41, 50], so that we always have $\delta < \Delta$. In practice, Δ provides a conservative estimate of the message delay during synchronous periods, which may be much higher than the maximal delay δ in a particular execution. We modify the protocol in Figure 4 so that in lines 7-8 it does not increase $\mathsf{dur\_recovery}$ and $\mathsf{dur\_delivery}$ above 6Δ and 4Δ, respectively. This corresponds to the bounds in Lemma 5 and preserves the protocol

liveness. Finally, we assume that periodic handlers (line 4 in Figure 3 and line 11 in Figure 4) are executed every ρ time units, and that the latency of reliable broadcast in line 31 under synchrony is $\leq \delta + \rho$ (this corresponds to an implementation that just periodically retransmits `DECISION` messages).

We quantify the latency of PBFT-light in both bad and good cases. For the bad case we assume that the protocol starts during the asynchronous period. Given a value x broadcast before GST, we quantify how quickly after GST all correct processes deliver x. For simplicity, we assume that timeouts are high enough at GST and that $\mathsf{leader}(\mathcal{V})$ is correct.

▶ **Theorem 8.** *Assume that before* GST *all correct processes start executing the protocol and one of them broadcasts* x*. Let* $\mathcal{V}$ *be defined as in Theorem 3 and assume that* $\mathsf{leader}(\mathcal{V})$ *is correct and at* GST *each correct process has* $\mathsf{dur\_recovery} > 6\delta$ *and* $\mathsf{dur\_delivery} > 4\delta$*. Then all correct processes deliver* x *by* $\mathsf{GST} + \rho + \max\{\rho + \delta, 6\Delta\} + 4\Delta + \max\{\rho, \delta\} + 7\delta$*.*

Although the latency bound looks complex, its main message is simple: PBFT-light recovers after a period of asynchrony in bounded time. This time is dominated by multiples of Δ; without the assumption that $\mathsf{leader}(\mathcal{V})$ is correct it would also be multiplied by f due to going over up to f views with faulty leaders. In [17, §C.3] we show the bound using the latency guarantees of our synchronizer (Properties A and B in Theorem 3).

We now consider the case when the protocol starts during the synchronous period, i.e., after GST. The following theorem quantifies how quickly all correct processes enter the first functional view, which in this case is view 1. If $\mathsf{leader}(1)$ is correct, it also quantifies how quickly a broadcast value x is delivered by all correct processes. The bound takes into account the following optimization: in view 1 the processes do not need to exchange `NEW_LEADER` messages. Then, after the systems starts up, the protocol delivers values within 4δ, which matches an existing lower bound of 3δ for the delivery time starting from the leader [5].

▶ **Theorem 9.** *Assume that all correct processes start the protocol after* GST *with* $\mathsf{dur\_recovery} > 5\delta$ *and* $\mathsf{dur\_delivery} > 4\delta$*. Then the* $\mathcal{V}$ *defined in Theorem 3 is equal to* 1 *and* $E_{\text{last}}(1) \leq T_{\text{last}}(0) + \delta$*. Furthermore, if a correct process broadcasts* x *at* $t \geq \mathsf{GST}$ *and* $\mathsf{leader}(1)$ *is correct, then all correct processes deliver* x *by* $\max\{t, T_{\text{last}}(0) + \delta\} + 4\delta$*.*

7 Additional Case Studies

To demonstrate the generality of SMR synchronizers, we have also used it to ensure the liveness of two other protocols. First, we handle a variant of PBFT that periodically forces a leader change, as is common in modern Byzantine SMR [28, 55, 56]. In this protocol a process calls `advance` not only when it suspects the current leader to be faulty, but also when it delivers B values proposed by this leader (for a fixed B). Second, we have applied the SMR synchronizer to a variant of the above protocol that follows the approach of HotStuff [57]. The resulting protocol adds an extra communication step to the normal path of PBFT in exchange for reducing the communication complexity of leader change. Due to space constraints, we defer the details about these two protocols to [17, §D] and [17, §E]. Their liveness proofs follow the methodology we proposed for PBFT-light, establishing analogs of Lemmas 4-6.

For PBFT with periodic leader rotation we have also established latency bounds when using the synchronizer in Figure 3 (see [17, §D]). The most interesting one (Theorem 56) demonstrates the benefit of PBFT's mechanism for adapting timeouts to an unknown δ: recall that in PBFT a process only increases its timeouts when a timer expires, which means that the current view does not operate normally (§4). We show that, since the protocol does

not increase its timeouts in good views (with correct leaders and under synchrony), it pays a minimal latency penalty to recover the first time it encounters a bad leader – the initial value of dur_recovery. This contrasts with the simplistic way of adapting the timeouts to an unknown δ by increasing them in every view: in this case, as the protocol keeps changing views, the timeouts would eventually increase up to the maximum (determined by Δ), and the protocol would have to wait that much to recover from a faulty leader.

8 Related Work and Discussion

Failure detectors. Failure detectors and leader oracles [26, 34] have been widely used for implementing consensus and SMR under benign failures [38, 39, 47], but their implementations under Byzantine failures are either impractical [43] or detect only restricted failure types [30, 40, 46]. Another approach was proposed in a textbook by Cachin et al. [20]. This relies on a leader-based Byzantine Epoch-Change (BEC) abstraction, which accepts "complain" hints from the application suggesting that the trust in the current leader should be revoked. However, like the classical leader oracles, BEC requires all correct processes to eventually trust the same correct leader, which is impossible to achieve in Byzantine settings. In fact, the BEC-based Byzantine consensus algorithm in §5.6.4 of [20] suffers from a liveness bug, which we describe in [17, §F]. The bug has been confirmed with one of the textbook's authors [19].

Although our `advance` is similar to "complain", we use it to implement a weaker abstraction of an SMR synchronizer. We then obtain properties similar to accuracy and completeness of failure detectors by carefully combining SMR-level timers with uses of `advance` (Lemmas 4-5). Also, while [20] does not specify constraints on the use of "complain" (see [17, §F]), we give a complete characterization of `advance` and show its sufficiency for solving SMR.

BFT-SMaRt [13, 54] built on the ideas of [20] to propose an abstraction of *validated and provable (VP) consensus*, which allows its clients to control leader changes. Although the overall BFT-SMaRt protocol appears to be correct, its liveness proof sketch suffers from issues with rigor similar to those of [20]. In particular, the conditions on how to change the leader in VP-Consensus to ensure its liveness were underspecified (again, see [17, §F]).

Emulating synchrony. Alternative abstractions avoid dependency on the specifics of a failure model by simulating synchrony [14, 27, 35, 42]. The first such abstraction is due to Awerbuch [10] who proposed a family of synchronizer algorithms emulating a round-based synchronous system of top of an asynchronous network with reliable communication and processes. The first such emulation in a failure-prone partially synchronous system was introduced in the DLS paper [32]. It relied on an expensive clock synchronization protocol, which interleaved its messages with every step of a high-level consensus algorithm implemented on top of it. Later work proposed more practical solutions, which reduce the synchronization frequency by relying on either timers [31] or synchronized hardware clocks [3, 7, 36] (the latter can be obtained using one of the existing fault-tolerant clock synchronization algorithms [29, 53]). However, the DLS model emulates communication-closed rounds, i.e., eventually, a process in a round r receives *all* messages sent by correct processes in r. This property rules out *optimistically responsive* [51, 57] protocols such as PBFT, which can make progress as soon as they receive messages from *any quorum*.

Consensus synchronizers. To address the shortcoming of DLS rounds, recent work proposed a more flexible abstraction ("consensus synchronizer" in §3) that switches processes through an infinite series of *views* [16, 49, 57]. In contrast to rounds, each view may subsume multiple

communication steps. Although consensus synchronizers can be used for efficient single-shot Byzantine consensus [16], using them for SMR results in suboptimal implementations. A classical approach is to decide on each SMR command using a separate black-box consensus instance [52]. However, implementing the latter using a consensus synchronizer would force the processes in every instance to iterate over the same sequence of potentially bad views until the one with a correct leader and sufficiently long duration could be reached.

An alternative approach was proposed in HotStuff [57]. This SMR protocol is driven by a *pacemaker*, which keeps generating views similarly to a consensus synchronizer. Within each view HotStuff runs a voting protocol that commits a block of client commands in a growing hash chain. Although the voting protocol is optimistically responsive, committing the next block is delayed until the pacemaker generates a new view, which increases latency. The cost the pacemaker may incur to generate a view is also paid for every single block.

SMR synchronizers. In contrast to the above approaches, SMR synchronizers allow the application to initiate view changes on demand via an `advance` call. As we show, this affords SMR protocols the flexibility to judiciously manage their view synchronization schedule: in particular, it prevents the timeouts from growing unnecessarily (§7) and avoids the overheads of further view synchronizations once a stable view is reached (Lemma 5, §5).

The first synchronizer with a `new_view`/`advance` interface, which here we call an SMR synchronizer, was proposed by Naor et al. [48,49]. They used it as an intermediate module in a communication-efficient implementation of a consensus synchronizer. The latter is sufficient to ensure the liveness of HotStuff [57] via either of the two straightforward SMR constructions we described above. The specification of the `new_view`/`advance` module of Naor et al. was only used as a stepping stone in the proof of their consensus synchronizer, and as a result, is more low-level and complex than our SMR synchronizer specification. Naor et al. did not investigate the usability of the SMR synchronizer abstraction as a generic building block applicable to a wide range of Byzantine SMR protocols – a gap we fill in this paper. Finally, they only handled a simplified version of partial synchrony where messages are never lost and δ is known a priori, whereas our SMR synchronizer implementation handles partial synchrony in its full generality. This implementation builds on the consensus synchronizer of Bravo et al. [16]. However, its correctness proof and performance analysis are more intricate, since the timing of the view switches is not fixed a priori, but driven by external `advance` inputs.

Aştefănoaei et al. [6] proposed another framework for implementing Byzantine SMR protocols, based on DLS rounds. This uses a simple synchronizer that does not exchange any messages: it recovers from a period of asynchrony by progressively increasing round durations until they are long enough for all correct processes to overlap in the same round. This way of view synchronization rules out optimistically responsive SMR protocols and does not bound the time to reach a decision after GST, as we do.

SMR liveness proofs. PBFT [23–25] is a seminal protocol whose design choices have been widely adopted [37, 44, 55, 56]. To the best of our knowledge, our proof in §5 is the first one to formally establish its liveness. An informal argument given in [25, §4.5.1] mainly justifies liveness assuming all correct processes enter a view with a correct leader and stay in that view for sufficiently long. It does not rigorously justify why such a view will be eventually reached, and in particular, how this is ensured by the interplay between SMR-level timeout management and view synchronization (§5). Liveness mechanisms were also omitted from the formal specification of PBFT by an I/O-automaton [23, 25].

Bravo et al. [16] have applied consensus synchronizers to several consensus protocols, including a single-shot version of PBFT. These protocols and their proofs and are much more straightforward than the full SMR protocols we consider here. In particular, since a consensus synchronizer keeps switching processes between views regardless of whether their leaders are correct, the proof of the single-shot PBFT in [16] does not need to establish analogs of completeness and accuracy (Lemmas 4 and 5) or deal with the fact that processes may disagree on timeout durations (Lemma 6).

Byzantine SMR protocols often integrate view synchronization into the core protocol, enabling white-box optimizations [1, 9, 18, 24]. Our work does not rule out this approach, but allows making it more systematic: we can first develop efficient mechanisms for view synchronization independently from SMR protocols, and do white-box optimizations afterwards.

References

1 DiemBFT v4: State machine replication in the Diem blockchain. URL: `https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf`.

2 Incorrect by construction-CBC Casper isn't live. URL: `https://derekhsorensen.com/docs/CBC_Casper_Flaw.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 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of Byzantine broadcast: a complete categorization. In *Symposium on Principles of Distributed Computing (PODC)*, 2021.

6 Lăcrămioara Aştefănoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci-Piergiovanni, and Eugen Zălinescu. Tenderbake – A solution to dynamic repeated consensus for blockchains. In *Symposium on Foundations and Applications of Blockchain (FAB)*, 2021.

7 Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Generating fast indulgent algorithms. In *International Conference on Distributed Computing and Networking (ICDCN)*, 2011.

8 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.

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

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 Christian Berger, Hans P. Reiser, and Alysson Bessani. Making reads in BFT state machine replication fast, linearizable, and live. In *Symposium on Reliable Distributed Systems (SRDS)*, 2021.

13 Alysson Neves Bessani, João Sousa, and Eduardo Adílio Pelinson Alchieri. State machine replication for the masses with BFT-SMART. In *Conference on Dependable Systems and Networks (DSN)*, 2014.

14 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.

15 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

16 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making Byzantine consensus live. In *Symposium on Distributed Computing (DISC)*, 2020.

17 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Liveness and latency of Byzantine state-machine replication (extended version). *arXiv*, abs/2202.06679, 2022. `arXiv:2202.06679`.

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

19 Christian Cachin. Personal communication, 2022.

20 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2 ed.)*. Springer, 2011.

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

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

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

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

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

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

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

28 Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

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

30 Assia Doudou, Benoît Garbinato, and Rachid Guerraoui. Abstractions for devising Byzantine-resilient state machine replication. In *Symposium on Reliable Distributed Systems (SRDS)*, 2000.

31 Cezara Dragoi, Josef Widder, and Damien Zufferey. Programming at the edge of synchrony. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020.

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

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

34 Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9:1–9:40, 2011.

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

36 Seth Gilbert, Rachid Guerraoui, and Dariusz R. Kowalski. On the message complexity of indulgent consensus. In *Symposium on Distributed Computing (DISC)*, 2007.

37 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.

38 Rachid Guerraoui. Indulgent algorithms (preliminary version). In *Symposium on Principles of Distributed Computing (PODC)*, 2000.

39 Rachid Guerraoui and Michel Raynal. The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453–466, 2004.

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

41 Amir Herzberg and Shay Kutten. Fast isolation of arbitrary forwarding faults. In *Symposium on Principles of Distributed Computing (PODC)*, 1989.

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

43 Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

44 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2010.

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

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

47 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.

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

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

50 Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *Symposium on Distributed Computing (DISC)*, 2017.

51 Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2018.

52 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

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

54 João Sousa. *Byzantine State Machine Replication for the Masses*. PhD thesis, University of Lisbon, 2017.

55 Chrysoula Stathakopoulou, Tudor David, and Marko Vukolić. Mir-BFT: High-throughput BFT for blockchains. *arXiv*, abs/1906.05552, 2019. `arXiv:1906.05552`.

56 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Symposium on Reliable Distributed Systems (SRDS)*, 2009.

57 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.