Distributed Computing (2022) 35:503–532
https://doi.org/10.1007/s00446-022-00432-y

Making Byzantine consensus live

Manuel Bravo[1] · Gregory Chockler[2] · Alexey Gotsman[1]

Received: 2 April 2021 / Accepted: 9 June 2022 / Published online: 2 September 2022

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 PBFT and HotStuff. 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.

Keywords Byzantine consensus · Blockchain · Partial synchrony · Liveness

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 [7,31] and open one [14,15,30], 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 [14,31,45] that improve on classical solutions, such as DLS [27] and PBFT [19].

Designing Byzantine consensus protocols is challenging, as witnessed by a number of bugs found in recent protocols [2,5,17,35]. Historically, researchers have paid more attention to safety of these protocols rather than liveness: e.g., while PBFT came with a safety proof [18], 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 [21], 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

This article is a revised and expanded version of a paper that appeared at the 2020 International Symposium on Distributed Computing (DISC).

✉ Alexey Gotsman
dev@null.com

1 IMDEA Software Institute, Madrid, Spain

2 University of Surrey, Guildford, UK

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 [14,19]. In contrast, both the seminal DLS work on consensus under partial synchrony [27] and some of the more recent work [1,41,45] 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. [1] 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. [41] 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 (Sect. 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 (Sect. 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 PBFT [19] (Sect. 5.1), HotStuff [45] (Sect. 5.2) and a two-phase version of the latter similar to Tendermint [14] (Sect. 5.3); in [12, §B] we also analyze 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 (Sect. 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. Our analysis stipulates an a priori known conservative message delay estimate Δ, which bounds the actual post-GST message delay δ in every execution. This allows us, for the first time, to derive closed-form expressions for Byzantine consensus latency bounds in a partial synchrony model where δ both is unknown and only holds 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 highlight the similarities and differences among protocols in a pedagogical fashion, thus providing a tutorial on modern Byzantine consensus.

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.

We consider a generalized *partial synchrony* model [21, 27], which guarantees that, 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, although for simplicity we assume that self-addressed messages are never lost and are delivered to the sender instantaneously. As in [21], 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. However, to state some of our latency bounds, we also assume the existence of a known upper bound Δ on the maximum value of δ in *any* execution [33]. 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. Finally, we 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). We denote the set of time points by Time (ranged over by t) and assume that local message processing takes zero time.

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 $\mathsf{View} = \{1, 2, \dots\}$ 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

1. $\forall i, v, v'.\, (E_i(v)$ and $E_i(v')$ are defined$) \implies (v < v' \iff E_i(v) < E_i(v'))$
2. $E_{\mathrm{first}}(\mathcal{V}) \geq \mathsf{GST}$
3. $\forall i.\, \forall v \geq \mathcal{V}.\, p_i$ is correct $\implies p_i$ enters v
4. $\forall v \geq \mathcal{V}.\, E_{\mathrm{last}}(v) \leq E_{\mathrm{first}}(v) + d$
5. $\forall v \geq \mathcal{V}.\, E_{\mathrm{first}}(v+1) > E_{\mathrm{first}}(v) + F(v)$

A. $\forall v \geq \mathcal{V}.\, E_{\mathrm{last}}(v+1) \leq E_{\mathrm{last}}(v) + F(v) + \delta$
B. $S_{\mathrm{first}} \geq \mathsf{GST} \wedge F(1) \geq 2\delta \implies \mathcal{V} = 1 \wedge E_{\mathrm{last}}(1) \leq S_{\mathrm{last}} + \delta$
C. $S_{\mathrm{first}} < \mathsf{GST} \wedge S_{f+1} \leq \mathsf{GST} + \rho \wedge F(\mathsf{GV}(\mathsf{GST} + \rho) + 1) \geq 2\delta \implies \mathcal{V} = \mathsf{GV}(\mathsf{GST} + \rho) + 1 \wedge E_{\mathrm{last}}(\mathcal{V}) \leq \mathsf{GST} + \rho + F(\mathcal{V} - 1) + 3\delta$

Fig. 2 Synchronizer properties (holding for some $\mathcal{V} \in \mathsf{View}$). Properties 1–5 specify the synchronizer abstraction, sufficient to ensure consensus liveness. Properties A–C give latency bounds specific to our FastSync synchronizer (Sect. 3.1). The latter satisfies Property 4 for $d = 2\delta$. The parameter ρ is the retransmission interval used by FastSync

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 for the protocol to terminate. To this end, the synchronizer is parameterized by a function defining this duration—$F : \mathsf{View} \cup \{0\} \to \mathsf{Time}$, which is monotone and such that $F(0) = 0$. The liveness of a protocol usually relies on F reaching a particular value u:

$$\exists v.\, \forall v'.\, v' \geq v \implies F(v') \geq u. \quad (1)$$

This can be satisfied, e.g., by letting $F(v) = 2v$ or $F(v) = 2^v$.

Properties 1–5 of Fig. 2 define our synchronizer specification, and Fig. 1 illustrates them visually. (We explain Properties A–C later; Property A is also illustrated in Fig. 1). 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 Sect. 5). On the other hand, it can be efficiently implemented

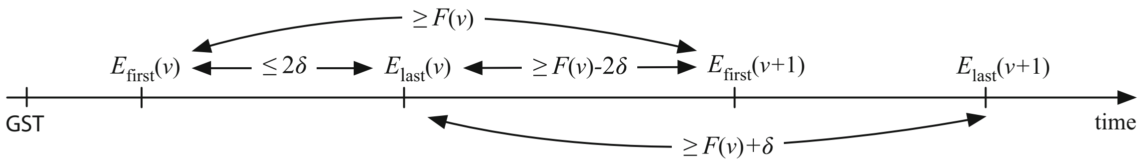

Fig. 1 Visual illustration of the synchronizer properties

under partial synchrony by our FastSync synchronizer (Sect. 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:

$$\begin{aligned}\forall v \geq \mathcal{V}.\ & E_{\text{first}}(v+1) - E_{\text{last}}(v) \\ & > (E_{\text{first}}(v) + F(v)) - (E_{\text{first}}(v) + d) \\ & = F(v) - d. \qquad (2)\end{aligned}$$

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 (due to (1) for a large enough u). 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 (Sect. 5.3) and Tendermint (Sect. 5.4).

3.1 FastSync: a bounded-space synchronizer for partial synchrony

In Algorithm 1 we present our FastSync synchronizer, which satisfies the synchronizer specification (Properties 1–5 of Fig. 2) 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 disseminates a special `WISH`(v') message, announcing its intention. Each process maintains an array max_views : $\{1, \dots, n\} \to$ 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 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 view $\leq$ view$^+$.

The process enters the view determined by the view variable (line 19) when the latter increases (view $>$ *prev_v* in line 16; we explain the extra condition later). At this point the process also resets its 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 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 view$^+$ = 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 view + 1 and view$^+$. In other words, 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$^+$.

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 pro-

```
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     $\mathsf{view} \leftarrow \max\{v \mid \exists k.\, \mathsf{max\_views}[k] = v \wedge$
               $|\{j \mid \mathsf{max\_views}[j] \geq v\}| \geq 2f+1\}$;
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, $F(\mathsf{view})$);
19         trigger new_view(view);
20     if $\mathsf{view}^+ > prev\_v^+$ then
21         send WISH($\mathsf{view}^+$) to all;
```

Algorithm 1: The FastSync synchronizer at a process p_i. The periodic handler is invoked every ρ units of time.

cess has already done so in line 21 due to receiving $f+1$ WISHes from other processes.

The guard $\mathsf{max\_views}[i] > 0$ in line 9 ensures that the first WISH sent by a process is always triggered by executing the code at line 3, 5, or 21. Since we assume that self-addressed messages are instantaneously delivered to the sender, this guard becomes permanently enabled as soon as the first WISH from any one of those lines has been sent.

Discussion FastSync requires only $O(n)$ variables for storing views. The algorithm also ensures that eventually every view is entered by all correct processes (Property 3), and views entered by correct processes throughout an execution do not skip values; the latter is formalized by the following lemma, proved in Sect. 4.1.

Lemma 1 *For all views v and v' such that $0 < v < v'$, if a correct process enters v', then some correct process has previously entered v.*

Thus, although the individual view values stored by FastSync are unbounded, Property 3 and Lemma 1 limit the power of the adversary to exhaust their allocated space, similarly to [9].

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 [11]. However, Bracha's algorithm only makes a step upon receiving a set of *identical* messages. Thus, its naive application to view synchronization [41, §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 WISH(v_2+1), and p_i sends WISH(v_2+2) (line 10). The processes in P_1 eventually receive the WISHes from $P_2 \cup \{p_i\}$, so that they set $\mathsf{view}^+ = v_2+1$ and send WISH(v_2+1) (line 21). Note that here processes act on $f+1$ mismatching WISHes, unlike in Bracha broadcast. Eventually, the processes in $P_1 \cup P_2$ receive $2f$ copies of WISH(v_2+1) and one 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 WISHes, again unlike in Bracha broadcast. Finally, the processes $P_1 \cup P_2$ time out and send WISH(v_2+2) (line 5), which allows all correct processes to enter v_2+2. Acting on sets of mismatching WISHes 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 Fig. 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 in Fig. 2 and will explain them shortly.

Theorem 1 *Consider an execution with an eventual message delay bound δ, and assume that (1) holds for $u = 2\delta$. Then there exists a view $\mathcal{V}$ such that in this execution* FastSync *satisfies all the properties in Fig. 2 for $d = 2\delta$.*

We prove this theorem in Sect. 4. An easy way to satisfy the premise of Theorem 1 in *any* execution is to pick a function F that grows without bound, so that it eventually reaches 2δ. However, practical implementations stop increasing timeouts once they exceed a reasonable value. We can reflect this in our model using the fact that $\delta \le \Delta$ in any execution. Thus, to ensure that (1) holds for $u = N\delta$ for some $N \ge 1$, it is enough to require that the output of F eventually becomes a constant $U = N\Delta$:

$$\exists v.\, \forall v'.\, v' \ge v \implies F(v') = U. \tag{3}$$

Proposition 1 *Let δ be the eventual message delay bound in some execution. Then, for all $N > 0$, if (3) holds for $U = N\Delta$, then (1) holds for $u = N\delta$.*

The above implies that the conclusion of Theorem 1 is satisfied provided (3) holds for $U = 2\Delta$, i.e., the output of F eventually becomes a known constant, and never changes afterwards.

We next describe the additional properties guaranteed by Theorem 1. Property A bounds the time for switching to the next view. This allows us to quantify the cost of switching between several views (e.g., due to faulty leaders), as formalized by the following proposition.

Proposition 2 *For any v and v' such that $\mathcal{V} \le v \le v'$,*

$$E_{\text{last}}(v') \le E_{\text{last}}(v) + \sum_{k=v}^{v'-1} (F(k) + \delta).$$

Proof Given Property 3, we prove the proposition by induction on views $v' \ge v$. The base case of $v' = v$ holds trivially. For the inductive step, assume that the desired inequality holds for $v' = w \ge v$. By Property A, $E_{\text{last}}(w + 1) \le E_{\text{last}}(w) + F(w) + \delta$. Combining this with the induction hypothesis, we get the desired inequality for $v' = w + 1$:

$$\begin{aligned} E_{\text{last}}(w+1) &\le E_{\text{last}}(w) + F(w) + \delta \\ &\le E_{\text{last}}(v) + \sum_{k=v}^{w-1} (F(k) + \delta) + F(w) + \delta \\ &\le E_{\text{last}}(v) + \sum_{k=v}^{w} (F(k) + \delta). \end{aligned}$$
□

Property B guarantees that, when the synchronizer starts after GST ($S_{\text{first}} \ge$ GST) and the initial timeout is long enough ($F(1) \ge 2\delta$), processes synchronize in the very first view ($\mathcal{V} = 1$) and enter it within δ of the last correct process calling `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 is at least 2δ, and $f + 1$ correct processes call `start()` early enough. By Proposition 1, we can apply Theorem 1 if (3) holds for $U = 2\Delta$. Then $F(\mathcal{V} - 1) \le 2\Delta$, and therefore, Property C implies $E_{\text{last}}(\mathcal{V}) \le \mathsf{GST} + \rho + 2\Delta + 3\delta$.

In Appendix A we also analyze the FastSync latency after GST under the assumption that the timeout of the first view entered by a correct process after GST is $\le 2\delta$. We use this result to establish latency bounds assuming exponentially growing timeouts, which is a common choice in practice (e.g., [19]). In particular, we show that all correct processes are guaranteed to enter synchronized view within $O(\delta \lg \delta)$ after S_{last}, if the protocol is started after GST, and within $O(\max\{\delta \lg \delta, \Delta\})$ after $\mathsf{GST} + \rho$, otherwise. The latter guarantees that the latency of view synchronization is bounded after GST.

4 Proof of FastSync correctness and latency bounds

To prove Theorem 1, in Sect. 4.1 we first prove that FastSync is a correct implementation of the synchronizer abstraction, as formalized by Properties 1–5 in Fig. 2. Then in Sect. 4.2 we prove the latency bounds stated by Properties A–C.

4.1 Proof of FastSync correctness

We now prove:

Theorem 2 *Consider an execution with an eventual message delay bound δ, and assume that (1) holds for $u = 2\delta$. Then there exists a view $\mathcal{V}$ such that in this execution* FASTSYNC *satisfies Properties 1–5 in Fig. 2 for $d = 2\delta$.*

To prove the theorem, we first introduce the following definitions. The *local view* of a process p_i at time t, denoted $\mathsf{LV}_i(t)$, is the latest view entered by p_i at or before t, or 0 if p_i has not entered any views by then. Thus, $\mathsf{GV}(t) = \max\{\mathsf{LV}_i(t) \mid p_i \text{ is correct}\}$. We say that a process p_i *attempts to advance* from a view $v \geq 0$ at time t if at this time p_i executes the code in either line 3 or line 5, and $\mathsf{LV}_i(t) = v$.

The next three lemmas establish basic constraints on the ordering of events generated in an execution of FASTSYNC. In particular, Lemma 2 shows that a $\mathtt{WISH}(v)$ message with $v > 0$ can only be sent by a correct process if some correct process has already attempted to advance from the view $v - 1$. Lemma 3 shows that a correct process can only enter a view $v > 0$ if some correct process has already attempted to advance from the view $v - 1$. Lemma 4 shows that some a correct process can only send a WISH message if some correct process has already called `start`.

Lemma 2 *For all times t and views $v > 0$, if a correct process sends $\mathtt{WISH}(v)$ at t, then there exists a time $t' \leq t$ such that some correct process attempts to advance from $v - 1$ at t'.*

Proof We first prove the following auxiliary proposition:

$$\forall p_i.\,\forall v.\ p_i \text{ is correct} \wedge p_i \text{ sends } \mathtt{WISH}(v) \text{ at } t \implies \\ \exists t' \leq t.\,\exists v' \geq v-1.\,\exists p_j.\ p_j \text{ is correct} \wedge \\ p_j \text{ attempts to advance from } v' \text{ at } t'. \tag{4}$$

By contradiction, assume that a correct process p_i sends $\mathtt{WISH}(v)$ at t, but for all $t' \leq t$ and all $v' \geq v - 1$, no correct process attempts to advance from v' at t'. Consider the earliest time t_k when some correct process p_k sends a $\mathtt{WISH}(v_k)$ with $v_k \geq v$, so that $t_k \leq t$. Then either $v_k = p_k.\mathsf{view}^+(t_k)$ or $p_k.\mathsf{view}(t_k) = p_k.\mathsf{view}^+(t_k) = v_k - 1$. If $p_k.\mathsf{view}^+(t_k) = v_k \geq v$, then $p_k.\mathsf{max\_views}(t_k)$ includes $f + 1$ entries $\geq v_k \geq v$, and therefore, there exists a correct process p_l that sent $\mathtt{WISH}(v')$ with $v' \geq v$ at $t_l < t_k$, contradicting the assumption that t_k is the earliest time when this can happen. Suppose that $p_k.\mathsf{view}(t_k) = p_k.\mathsf{view}^+(t_k) = v_k - 1$. Then at t_k the process p_k executes either line 5 or line 10 and $\mathsf{LV}_k(t_k) = v_k - 1$. If p_k executes line 5 at t_k, then since $\mathsf{LV}_k(t_k) = v_k - 1$, p_k attempts to advance from $v_k - 1 \geq v - 1$ at $t_k \leq t$, contradicting our assumption that no such attempt can occur.

Suppose now that p_k executes the code in line 10 at t_k and $p_k.\mathsf{view}(t_k) = p_k.\mathsf{view}^+(t_k) = v_k - 1$. Consider first the case when $v_k = 1$. Since $\mathsf{max\_views}[k] > 0$, p_k has already sent $\mathtt{WISH}(v'_k)$ for some view $v'_k \geq 1$ at a time $< t_k$. Since $v'_k \geq v_k \geq v$, this is a contradiction to our assumption that no WISH messages with views $\geq v$ can be sent before t_k. It remains to consider the case when $v_k > 1$. Then $E_k(v_k - 1)$ is defined and satisfies $E_k(v_k - 1) < t_k$. Thus, $p_k.\mathsf{view}(E_k(v_k - 1)) = p_k.\mathsf{view}^+(E_k(v_k - 1)) = v_k - 1$. Since p_k starts $p_k.\mathsf{timer\_view}$ at $E_k(v_k - 1)$, and $p_k.\mathsf{timer\_view}(t_k)$ is not enabled, there exists a time t'_k such that $E_k(v_k - 1) < t'_k < t_k$ and $p_k.\mathsf{timer\_view}$ expires at t'_k, triggering the execution of the $\mathsf{timer\_view}$ expiration handler. Since both $p_k.\mathsf{view}$ and $p_k.\mathsf{view}^+$ are non-decreasing, and both are equal to $v_k - 1$ at $E_k(v_k - 1)$ as well as t_k, $p_k.\mathsf{view}(t'_k) = p_k.\mathsf{view}^+(t'_k) = v_k - 1$. Thus, $\mathsf{LV}_k(t'_k) = v_k - 1$, which implies that at $t'_k < t_k \leq t$, p_k attempts to advance from $v_k - 1 \geq v - 1$, contradicting our assumption that no such attempt can happen. We conclude that (4) holds.

We now prove the lemma. Let t and v be such that some correct process sends $\mathtt{WISH}(v)$ at t. By (4), there exists a correct process that attempts to advance from a view $\geq v - 1$ at or before t. Let t' be the earliest time when some correct process attempts to advance from a view $\geq v - 1$, and let p_j be this process and $v' \geq v - 1$ be the view from which p_j attempts to advance at t'. Thus, at t', p_j executes the code in either line 3 or line 5, and $\mathsf{LV}_j(t') = v' \geq v - 1$. Suppose first that p_j executes the code in line 5 at t'. Since $\mathsf{LV}_j(t') = v'$, there exists an earlier time at which $p_j.\mathsf{view}^+ = p_j.\mathsf{view} = v'$. Since $p_j.\mathsf{view}^+$ is non-decreasing, $p_j.\mathsf{view}^+(t') \geq v'$. If $p_j.\mathsf{view}^+(t') > v'$, then given that $v' \geq v - 1$, $p_j.\mathsf{view}^+(t') \geq v$. Thus, there exists a correct process p_k and time $t'' < t'$ such that p_k sent $\mathtt{WISH}(v'')$ with $v'' \geq v$ to p_j at t''. By (4), there exists a time $\leq t'' < t'$ at which some correct process attempts to advance from a view $\geq v'' - 1 \geq v - 1$, which is impossible. Thus, $p_j.\mathsf{view}^+(t') = v'$. Since $\mathsf{LV}_j(t') = v'$, we have $p_j.\mathsf{view}(t') = p_j.\mathsf{view}^+(t') = v'$. Suppose now that p_j executes the code in line 3. Then $p_j.\mathsf{view}^+(t') = p_j.\mathsf{view}(t') = 0 = \mathsf{LV}_j(t') = v'$. Hence, in both cases

$$p_j.\mathsf{view}(t') = p_j.\mathsf{view}^+(t') = v' \geq v - 1.$$

By the definitions of view and view^+, v' is both the lowest view among the highest $2f + 1$ views in $p_j.\mathsf{max\_views}(t')$, and the lowest view among the highest $f + 1$ views in $p_j.\mathsf{max\_views}(t')$. Hence, $p_j.\mathsf{max\_views}(t')$ includes $f + 1$ entries equal to v', and therefore, there exists a correct process p_k such that

$$p_j.\mathsf{view}(t') = p_j.\mathsf{view}^+(t') \\ = p_j.\mathsf{max\_views}[k](t') = v' \geq v - 1. \tag{5}$$

Also, for all correct processes p_l, $p_j.\mathsf{max\_views}[l](t') < v$: otherwise, some correct process sent $\mathtt{WISH}(v'')$ with $v'' \geq v$ at a time $< t'$, and therefore, by (4), some correct process attempted to advance from a view $\geq v - 1$ earlier than t', which is impossible. Thus,

$p_j.\mathsf{view}(t') = p_j.\mathsf{view}^+(t') = p_j.\mathsf{max\_views}[k](t') < v.$

Together with (5), this implies

$p_j.\mathsf{view}(t') = p_j.\mathsf{view}^+(t') = v - 1.$

Hence, $\mathsf{LV}_j(t') = v-1$, and therefore, p_j attempts to advance from $v-1$ at t'. Thus, $v' = v-1$ and $t' \leq t$, as required. □

Lemma 3 *If a correct process p_i enters a view v, then there exists a time $t < E_i(v)$ at which some correct process attempts to advance from $v-1$.*

Proof Since p_i enters a view v, we have $p_i.\mathsf{view}(E_i(v)) = p_i.\mathsf{view}^+(E_i(v)) = v$. By the definitions of view and view^+, v is both the lowest view among the highest $2f+1$ views in $p_i.\mathsf{max\_views}(E_i(v))$, and the lowest view among the highest $f+1$ views in $p_i.\mathsf{max\_views}(E_i(v))$. Hence, $p_i.\mathsf{max\_views}(E_i(v))$ includes $f+1$ entries equal to v. Then there exists a time $t' < E_i(v)$ at which some correct process sends $\mathtt{WISH}(v)$. Hence, by Lemma 2, there exists a time $t \leq t' < E_i(v)$ at which some correct process attempts to advance from $v-1$. □

Lemma 4 *For all times t and views $v > 0$, if a correct process sends $\mathtt{WISH}(v)$ at t, then there exists a time $t' \leq t$ such that some correct process calls $\mathtt{start}$ at t'.*

Proof Consider the earliest time $t_k \leq t$ at which some correct process p_k sends $\mathtt{WISH}(v_k)$ for some view v_k. By Lemma 2, there exists a time $t_j \leq t_k$ at which some correct process attempts to advance from $v_k - 1 \geq 0$, and therefore, sends $\mathtt{WISH}(v_k)$ at t_j. Since t_k is the earliest time when this could happen, we have $t_j = t_k$. Also, if $v_k - 1 > 0$, then $E_k(v_k-1)$ is defined, and hence, by Lemma 3, some correct process attempts to advance from $v_k - 2$ by sending $\mathtt{WISH}(v_k - 1)$ earlier than $t_j = t_k$, which cannot happen. Thus, at t_k, p_k attempts to advance from view 0, so that $v_k = 1$ and $\mathsf{LV}_k(t_k) = 0$. Assume first that p_k executes the code in line 5 at t_k. Then $p_k.\mathsf{timer\_view}$ expires at t_k, and hence, there exists a time $s_k < t_k$ such that $p_k.\mathsf{timer\_view}$ is set at s_k. Thus, at s_k, p_k enters a view > 0. Since LV is non-decreasing, $\mathsf{LV}_k(t_k) > 0$, which is a contradiction. Thus, p_k cannot execute line 5 at t_k, and has to call $\mathtt{start}$ at this time. □

By Lemma 3, no correct process can enter a view before some correct process sends a $\mathtt{WISH}$ message. Since, by Lemma 4, this cannot happen before some correct process calls $\mathtt{start}$, we have

Corollary 1 $\forall v.\, S_{\mathrm{first}} < E_{\mathrm{first}}(v)$.

We now prove that the views entered by correct processes throughout an execution of FastSync do not skip values, as stipulated by Lemma 1 (Sect. 3.1).

Proof of Lemma 1. Fix v' such that $v' \geq 2$ and assume that a correct process enters v', so that $E_{\mathrm{first}}(v')$ is defined. We prove by induction on k that for each $k = 0..(v'-1)$ some correct enters $v'-k$ no later than $E_{\mathrm{first}}(v')$. The base case of $k=0$ is trivial. For the inductive step, assume that the required holds for some k, so that $E_{\mathrm{first}}(v'-k) \leq E_{\mathrm{first}}(v')$. Then by Lemma 3, there exists a time $t < E_{\mathrm{first}}(v'-k)$ at which some correct process p_j attempts to advance from $v'-k-1$. But then p_j's local view at t is $v'-k-1$. Hence, p_j enters $v'-k-1$ before $t < E_{\mathrm{first}}(v'-k) \leq E_{\mathrm{first}}(v')$, as required. □

Lemma 5 below establishes that the views sent in the $\mathtt{WISH}$ messages by the same process can only increase. Its proof relies on the next proposition, stating a few simple invariants that follow immediately from the structure of the code and our assumption that every message sent by a process is instantaneously delivered to the sender (Sect. 3).

Proposition 3 *Let p_i be a correct process. The following conditions hold at all times in every execution of* FastSync*:*

1. $\forall v. \forall t.\ p_i$ *sends* $\mathtt{WISH}(v)$ *at* $t \implies v \in \{p_i.\mathsf{view}^+(t),\, p_i.\mathsf{view}^+(t)+1\}$.
2. $\forall v. \forall t.\ p_i$ *sends* $\mathtt{WISH}(v)$ *at* $t \wedge v = p_i.\mathsf{view}^+(t) + 1 \implies p_i.\mathsf{view}^+(t) = p_i.\mathsf{view}(t) \wedge (p_i.\mathsf{timer\_view}$ *is disabled*).
3. $\forall v. \forall t.\ p_i$ *sends* $\mathtt{WISH}(v)$ *at* $t \implies \forall t' > t.\ p_i.\mathsf{max\_views}[i](t') > 0 \wedge p_i.\mathsf{view}^+(t') > 0$.

Lemma 5 *For all views $v, v' > 0$, if a correct process sends $\mathtt{WISH}(v)$ before sending $\mathtt{WISH}(v')$, then $v \leq v'$.*

Proof Let s and s' such that $s < s'$ be the times at which a correct process p_i sends $\mathtt{WISH}(v)$ and $\mathtt{WISH}(v')$ messages, respectively. We show that $v' \geq v$. By Proposition 3(1), $v \in \{p_i.\mathsf{view}^+(s), p_i.\mathsf{view}^+(s)+1\}$ and $v' \in \{p_i.\mathsf{view}^+(s'), p_i.\mathsf{view}^+(s')+1\}$. Hence, if $v = p_i.\mathsf{view}^+(s)$ or $v' = p_i.\mathsf{view}^+(s')+1$, then we get $v \leq v'$ from the fact that $p_i.\mathsf{view}^+$ is non-decreasing. It thus remains to consider the case when $v = p_i.\mathsf{view}^+(s)+1$ and $v' = p_i.\mathsf{view}^+(s')$. In this case by Proposition 3(2), $p_i.\mathsf{view}^+(s) = p_i.\mathsf{view}(s)$ and $p_i.\mathsf{timer\_view}(s)$ is disabled. We now consider several cases depending on the line at which $\mathtt{WISH}(v')$ is sent.

– $\mathtt{WISH}(v')$ is sent at line 3. In this case we have $v' = 1$ and $p_i.\mathsf{view}^+(s') = 0$, so that $v' = p_i.\mathsf{view}^+(s')+1$. But this contradicts the assumption that $v' = p_i.\mathsf{view}^+(s')$, and thus this case is impossible.
– $\mathtt{WISH}(v')$ is sent at lines 5 or 10. Then $v' = p_i.\mathsf{view}^+(s') = \max(p_i.\mathsf{view}(s')+1,\, p_i.\mathsf{view}^+(s'))$. Since $p_i.\mathsf{view}$ is non-decreasing, we get $p_i.\mathsf{view}^+(s') \geq p_i.\mathsf{view}(s') +$

$1 > p_i.\mathsf{view}(s') \geq p_i.\mathsf{view}(s) = p_i.\mathsf{view}^+(s)$. Hence, $p_i.\mathsf{view}^+(s') > p_i.\mathsf{view}^+(s)$, and therefore, $v' = p_i.\mathsf{view}^+(s') \geq p_i.\mathsf{view}^+(s) + 1 = v$, as required.

– $\mathtt{WISH}(v')$ is sent at line 8. Then $p_i.\mathsf{timer\_view}(s')$ is enabled. Since $p_i.\mathsf{timer\_view}(s)$ is disabled, there exists a time s'' such that $s < s'' < s'$ and p_i enters a view at s''. By the view entry condition $p_i.\mathsf{view}(s'') > p_i.\mathit{prev\_v}(s'')$. Since $p_i.\mathsf{view}$ is non-decreasing, we get $p_i.\mathsf{view}^+(s') \geq p_i.\mathsf{view}(s') \geq p_i.\mathsf{view}(s'') > p_i.\mathsf{view}(s) = p_i.\mathsf{view}^+(s)$. Thus, $p_i.\mathsf{view}^+(s') > p_i.\mathsf{view}^+(s)$ and therefore, $v' = p_i.\mathsf{view}^+(s') \geq p_i.\mathsf{view}^+(s) + 1 = v$, as required.

– $\mathtt{WISH}(v')$ is sent at line 21. Then $p_i.\mathsf{view}^+(s') > p_i.\mathit{prev\_v}^+(s') \geq p_i.\mathsf{view}^+(s)$, and therefore, $v' = p_i.\mathsf{view}^+(s') \geq p_i.\mathsf{view}^+(s) + 1 = v$, as required. □

In order to cope with message loss before GST, every correct process retransmits the highest WISH it sent every ρ units of time, according to its local clock (lines 6–10). Eventually, one of these retransmissions will occur after GST, and therefore, there exists a time by which all correct processes are guaranteed to send their highest WISHes at least once after GST. The earliest such time, $\overline{\mathsf{GST}}$, is defined as follows:

$$\overline{\mathsf{GST}} = \begin{cases} \mathsf{GST} + \rho, & \text{if } S_{\text{first}} < \mathsf{GST}; \\ S_{\text{first}}, & \text{otherwise.} \end{cases}$$

From this definition it follows that

$$\overline{\mathsf{GST}} \geq \mathsf{GST}. \tag{6}$$

Lemma 6 below formalizes the key property of $\overline{\mathsf{GST}}$.

Lemma 6 *For all correct processes p_i, times $t \geq \overline{\mathsf{GST}}$, and views v, if p_i sends $\mathtt{WISH}(v)$ at a time $\leq t$, then there exists a view $v' \geq v$ and a time t' such that $\mathsf{GST} \leq t' \leq t$ and p_i sends $\mathtt{WISH}(v')$ at t'.*

Proof Let $s \leq t$ be the time at which p_i sends $\mathtt{WISH}(v)$. We consider two cases. Suppose first that $S_{\text{first}} \geq \mathsf{GST}$. By Lemma 4, $s \geq S_{\text{first}}$, and therefore, $\mathsf{GST} \leq s \leq t$. Thus, choosing $t' = s$ and $v' = v$ validates the lemma. Suppose next that $S_{\text{first}} < \mathsf{GST}$. Then by the definition of $\overline{\mathsf{GST}}$, $t \geq \mathsf{GST} + \rho$. If $s \geq \mathsf{GST}$, then $\mathsf{GST} \leq s \leq t$, and therefore, choosing $t' = s$ and $v' = v$ validates the lemma. Assume now that $s < \mathsf{GST}$. Since after GST the p_i's local clock advances at the same rate as real time, there exists a time s' satisfying $\mathsf{GST} \leq s' \leq t$ such that p_i executes the periodic retransmission code in lines 6–10 at s'. Since p_i already sent a WISH message at $s < \mathsf{GST} \leq s'$, by Proposition 3(3), $p_i.\mathsf{max\_views}[i](s') > 0$, and therefore, the code sending a WISH message is guaranteed to be reached at s'. Thus, there exists v' such that p_i sends $\mathtt{WISH}(v')$ at s' by executing the code in either line 8 or line 10, and $\mathsf{GST} \leq s' \leq t$. By Lemma 5, $v' \geq v$, which implies the required. □

We next state several lemmas that encapsulate the arguments showing the various properties in Fig. 2. The following lemma is used to prove Property 5.

Lemma 7 *If a correct process enters a view $v > 0$ and $E_{\text{first}}(v) \geq \mathsf{GST}$, then for all $v' > v$, no correct process attempts to advance from $v' - 1$ before $E_{\text{first}}(v) + F(v)$.*

Proof Suppose by contradiction that there exists a time $t' < E_{\text{first}}(v) + F(v)$ and a correct process p_i such that p_i attempts to advance from $v' - 1 > v - 1$ at t'. If p_i executes the code in line 3 at t', then $\mathsf{LV}_i(t') = 0 = v' - 1 > v - 1 \geq 0$, which is impossible. Thus, at t' the process p_i executes the code in line 5, and $\mathsf{LV}_i(t') = v' - 1$. Since $p_i.\mathsf{timer\_view}$ is not enabled at t', p_i must have entered $v' - 1$ at least $F(v)$ before t' according to its local clock. Since $v' - 1 \geq v$, by Lemma 1 we have $E_{\text{first}}(v' - 1) \geq E_{\text{first}}(v) \geq \mathsf{GST}$. Therefore, given that the clocks of all correct processes progress at the same rate as real time after GST, we get

$$E_{\text{first}}(v) \leq E_{\text{first}}(v' - 1) \leq t' - F(v' - 1).$$

Hence,

$$t' \geq E_{\text{first}}(v) + F(v' - 1).$$

Since F is non-decreasing and $v' - 1 \geq v$, we have $F(v' - 1) \geq F(v)$, so that

$$t' \geq E_{\text{first}}(v) + F(v),$$

which contradicts our assumption that $t' < E_{\text{first}}(v) + F(v)$. This contradiction shows the required. □

Corollary 2 *Consider a view v and assume that v is entered by a correct process. If $E_{\text{first}}(v) \geq \mathsf{GST}$, then a correct process cannot send a $\mathtt{WISH}(v')$ with $v' > v$ earlier than $E_{\text{first}}(v) + F(v)$.*

Proof Assume a correct process sends a $\mathtt{WISH}(v')$ with $v' > v$ at time t'. By Lemma 2, there exists a time $s \leq t'$ such that some correct process p_i attempts to advance from $v' - 1 > v - 1$ at s. By Lemma 7, $s \geq E_{\text{first}}(v) + F(v)$, which implies that $t' \geq s \geq E_{\text{first}}(v) + F(v)$, as required. □

The following lemma is used to prove Property 4 for $d = 2\delta$.

Lemma 8 *Consider a view $v > 0$ and assume that v is entered by a correct process. If $E_{\text{first}}(v) \geq \overline{\mathsf{GST}}$ and $F(v) \geq 2\delta$, then all correct processes enter v and $E_{\text{last}}(v) \leq E_{\text{first}}(v) + 2\delta$.*

Proof Since $E_{\text{first}}(v) \geq \overline{\mathsf{GST}}$, by (6), $E_{\text{first}}(v) \geq \mathsf{GST}$. Since $F(v) \geq 2\delta$, Corollary 2 implies that no correct process can send $\mathtt{WISH}(v')$ with $v' > v$ earlier than $E_{\text{first}}(v) + 2\delta$. Once any such $\mathtt{WISH}(v')$ is sent, it will take a non-zero time until it is received by any correct process. Thus, we have:

(*) no correct process receives $\mathtt{WISH}(v')$ with $v' > v$ from a correct process until after $E_{\text{first}}(v) + 2\delta$.

Let p_i be a correct process that enters v at $E_{\text{first}}(v)$. By the view entry condition, $p_i.\mathsf{view}(E_{\text{first}}(v)) = v$, and therefore $p_i.\mathsf{max\_views}(E_{\text{first}}(v))$ includes $2f + 1$ entries $\geq v$. At least $f + 1$ of these entries belong to correct processes, and by (*), none of them can be $> v$. Hence, there exists a set C of $f + 1$ correct processes, each of which sends $\mathtt{WISH}(v)$ to all processes before $E_{\text{first}}(v)$.

Since $E_{\text{first}}(v) \geq \overline{\mathsf{GST}}$, by Lemma 6, every $p_j \in C$ also sends $\mathtt{WISH}(v')$ with $v' \geq v$ at some time s_j such that $\mathsf{GST} \leq s_j \leq E_{\text{first}}(v)$. Then by (*) we have $v' = v$. It follows that each $p_j \in C$ is guaranteed to send $\mathtt{WISH}(v)$ to all correct processes between GST and $E_{\text{first}}(v)$. Since all messages sent by correct processes after GST are guaranteed to be received by all correct processes within δ of their transmission, by $E_{\text{first}}(v) + \delta$ all correct processes will receive $\mathtt{WISH}(v)$ from at least $f + 1$ correct processes.

Consider an arbitrary correct process p_j and let $t_j \leq E_{\text{first}}(v) + \delta$ be the earliest time by which p_j receives $\mathtt{WISH}(v)$ from $f+1$ correct processes. By (*), no correct process sends $\mathtt{WISH}(v')$ with $v' > v$ before $t_j < E_{\text{first}}(v) + 2\delta$. Thus, $p_j.\mathsf{max\_views}(t_j)$ includes at least $f+1$ entries equal to v and at most f entries $> v$, so that $p_j.\mathsf{view}^+(t_j) = v$. Then p_j sends $\mathtt{WISH}(v)$ to all processes no later than $t_j \leq E_{\text{first}}(v) + \delta$. Since $E_{\text{first}}(v) \geq \overline{\mathsf{GST}}$, by Lemma 6, p_j also sends $\mathtt{WISH}(v')$ with $v' \geq v$ in-between GST and $E_{\text{first}}(v)+\delta$. By (*), $v' = v$, and therefore, p_j sends $\mathtt{WISH}(v)$ to all processes sometime between GST and $E_{\text{first}}(v) + \delta$. Hence, all correct processes are guaranteed to send $\mathtt{WISH}(v)$ to all correct processes between GST and $E_{\text{first}}(v) + \delta$.

Consider an arbitrary correct process p_k and let $t_k \leq E_{\text{first}}(v) + 2\delta$ be the earliest time by which p_k receives $\mathtt{WISH}(v)$ from all correct processes. Then by (*), all entries of correct processes in $p_k.\mathsf{max\_views}(t_k)$ are equal to v. Since there are at least $2f + 1$ correct processes: *(i)* at least $2f + 1$ entries in $p_k.\mathsf{max\_views}(t_k)$ are equal to v, and *(ii)* one of the $f + 1$ highest entries in $p_k.\mathsf{max\_views}(t_k)$ is equal to v. From *(i)*, $p_k.\mathsf{view}^+(t_k) \geq p_k.\mathsf{view}(t_k) \geq v$, and from *(ii)*, $p_k.\mathsf{view}(t_k) \leq p_k.\mathsf{view}^+(t_k) \leq v$. Therefore, $p_k.\mathsf{view}(t_k) = p_k.\mathsf{view}^+(t_k) = v$, so that p_k enters v no later than $t_k \leq E_{\text{first}}(v) + 2\delta$. We have thus shown that by $E_{\text{first}}(v) + 2\delta$, all correct processes enter v, as required. □

The following lemma shows that processes keep entering new views forever. This is used to prove Property 3.

Lemma 9 *For all views v, there exists a view $v' > v$ such that some correct process eventually enters v'.*

Proof Assume by contradiction that the required does not hold and let v be the maximal view entered by a correct process; if there are no such views, we let $v = 0$. Thus, we have

$$\forall t.\, \forall v' \geq v+1.\, \forall p_i.\, \neg(p_i \text{ enters } v' \text{ at } t \wedge p_i \text{ is correct}). \quad (7)$$

If there is a correct process that sends $\mathtt{WISH}(v')$ with $v' > v + 1$ at any time s, then by Lemma 2, a correct process p_i attempts to advance from $v' - 1 > v$ at some time $s' \leq s$. Thus, $\mathsf{LV}_i(s') = v' - 1 \geq v + 1 \geq 1$, which contradicts (7). Thus, we have

$$\forall t.\, \forall v' > v + 1.\, \forall p_i.\, \neg(p_i \text{ sends } \mathtt{WISH}(v') \text{ at } t \wedge p_i \text{ is correct}). \quad (8)$$

Since we assume that all correct processes eventually call $\mathtt{start}$, there exists a time $T_1 = \max\{E_{\text{first}}(v), \overline{\mathsf{GST}}, S_{\text{last}}\}$. We consider two cases.

Suppose first that $v = 0$, so that no correct process enters any view. Consider a correct process p_i. This process must call $\mathtt{start}$ at some time $t_i \leq T_1$. If $p_i.\mathsf{view}^+ = 0$ at t_i, then p_i sends $\mathtt{WISH}(1)$ at t_i. On the other hand, if $p_i.\mathsf{view}^+ > 0$ at t_i, then p_i has already sent $\mathtt{WISH}(v')$ with $v' = p_i.\mathsf{view}^+ > 0$ when $p_i.\mathsf{view}^+$ first became > 0 at some time before t_i. By (8), $v' = 1$. Thus, in both cases, there exists a time $t_i' \leq T_1$ such that p_i sends $\mathtt{WISH}(1)$ at t_i'. Since $T_1 \geq \overline{\mathsf{GST}}$, by Lemma 6, there exists a view $v'' \geq 1$ and a time s_i such that $\mathsf{GST} \leq s_i \leq T_1$ and p_i sends $\mathtt{WISH}(v'')$ at s_i. By (8), $v'' = 1$. Since the links are reliable after GST, $\mathtt{WISH}(1)$ sent by p_i at s_i will be received by all correct processes. Thus, there exists a time $T_2 \geq T_1$ and a correct process p_j such that p_j receives $\mathtt{WISH}(1)$ from all correct processes at $t_j \leq T_2$. By (8), all entries of correct processes in $p_j.\mathsf{max\_views}(t_j)$ are equal to 1. Since there are at least $2f+1$ correct processes: *(i)* at least $2f + 1$ entries in $p_l.\mathsf{max\_views}(t_l)$ are equal to 1, and *(ii)* one of the $f+1$ highest entries in $p_l.\mathsf{max\_views}(t_l)$ is equal to 1. From *(i)*, $p_l.\mathsf{view}^+(t_l) \geq p_l.\mathsf{view}(t_l) \geq 1$, and from *(ii)*, $p_l.\mathsf{view}(t_l) \leq p_l.\mathsf{view}^+(t_l) \leq 1$. Hence, $p_l.\mathsf{view}(t_l) = p_l.\mathsf{view}^+(t_l) = 1$, and therefore, p_j enters view 1 at t_j, contradicting (7).

Next, suppose $v > 0$. Then some correct process entered v, and thus there exists a set C consisting of $f + 1$ correct processes all of which sent $\mathtt{WISH}(v')$ with $v' \geq v$ before T_1. Consider $p_i \in C$ and let $t_i \leq T_1$ be a time such that at t_i the process p_i sends $\mathtt{WISH}(v_i)$ with $v_i \geq v$. Since $T_1 \geq \overline{\mathsf{GST}}$, by Lemma 6, there exists a view $v_i' \geq v_i$ and a time s_i such that $\mathsf{GST} \leq s_i \leq T_1$ and p_i sends $\mathtt{WISH}(v_i')$ at s_i. By (8), we have $v_i' \in \{v, v + 1\}$. Since the links are reliable after GST, the $\texttt{WISH}(v_i')$ sent by p_i at s_i will be received by all correct processes.

Thus, there exists a time $T_2 \geq T_1 \geq \overline{\mathsf{GST}}$ by which all correct processes have received $\texttt{WISH}(v')$ with $v' \in \{v, v+1\}$ from all processes in C. Consider an arbitrary correct process p_j. By (8), the entry of every process in C in $p_j.\mathsf{max\_views}(T_2)$ is equal to either v or $v+1$. Since $|C| \geq f+1$ and all processes in C are correct, $p_j.\mathsf{max\_views}(T_2)$ includes at least $f+1$ entries $\geq v$. Thus, $p_j.\mathsf{view}^+(T_2) \geq v$, and therefore, p_j sends $\texttt{WISH}(v_j)$ with $v_j \geq v$ no later than at T_2. Since $T_2 \geq \overline{\mathsf{GST}}$, by Lemma 6, there exists a view $v_j' \geq v_j$ and a time s_j such that $\mathsf{GST} \leq s_j \leq t_j$ and p_j sends $\texttt{WISH}(v_j')$ at s_j. By (8), $v_j' \in \{v, v+1\}$. Since the links are reliable after GST, the $\texttt{WISH}(v_j')$ sent by p_j at s_j will be received by all correct processes.

Thus, there exists a time $T_3 \geq T_2 \geq \overline{\mathsf{GST}}$ by which all correct processes have received $\texttt{WISH}(v')$ with $v' \in \{v, v+1\}$ from all correct processes. Consider an arbitrary correct process p_k. Then at T_3, all entries of correct processes in $p_k.\mathsf{max\_views}$ are $\geq v$. By (8), each of these entries is equal to either v or $v+1$. Since at least $2f+1$ processes are correct: *(i)* at least $2f+1$ entries in $p_l.\mathsf{max\_views}(T_3)$ are $\geq v$, and *(ii)* one of the $f+1$ highest entries in $p_k.\mathsf{max\_views}(T_3)$ is $\leq v+1$. From *(i)*, $p_k.\mathsf{view}^+(T_3) \geq p_k.\mathsf{view}(T_3) \geq v$, and from *(ii)*, $p_k.\mathsf{view}(T_3) \leq p_k.\mathsf{view}^+(T_3) \leq v+1$. Hence, $p_k.\mathsf{view}(T_3)$, $p_k.\mathsf{view}^+(T_3) \in \{v, v+1\}$. Since no correct process enters $v+1$, $p_k.\mathsf{view}(T_3)$ and $p_k.\mathsf{view}^+(T_3)$ cannot be both simultaneously equal to $v+1$. Thus, $p_k.\mathsf{view}(T_3) = v$, and either $p_k.\mathsf{view}^+(T_3) = v$ or $p_k.\mathsf{view}^+(T_3) = v+1$. If $p_k.\mathsf{view}^+(T_3) = v+1$, then p_k has sent $\texttt{WISH}(v_k)$ with $v_k = v+1$ when $p_k.\mathsf{view}^+$ has first become equal to $v+1$ sometime before T_3. On the other hand, if $p_k.\mathsf{view}(T_3) = p_k.\mathsf{view}^+(T_3) = v$, then p_k has entered v and started $p_k.\mathsf{timer\_view}$ at or before T_3. Since p_k does not enter any higher views, $p_k.\mathsf{timer\_view}$ will eventually expire, causing p_k to send $\texttt{WISH}(v_k)$ with $v_k > v$. By (8), $v_k = v+1$. Thus, there exists a time $t_k \geq T_3$ by which p_k sends $\texttt{WISH}(v+1)$ to all processes. Since $t_k \geq T_3 \geq \overline{\mathsf{GST}}$, by Lemma 6, there exists a view $v_k' \geq v+1$ and a time s_k such that $\mathsf{GST} \leq s_k \leq T_3$ and p_k sends $\texttt{WISH}(v_k')$ at s_k. By (8), $v_k' = v+1$. Since the links are reliable after GST, the $\texttt{WISH}(v+1)$ sent by p_k will be received by all correct processes.

Thus, there exists a time $T_4 \geq T_3 \geq \overline{\mathsf{GST}}$ by which all correct processes have received $\texttt{WISH}(v+1)$ from all correct processes. Fix an arbitrary correct process p_l. By (8), all entries of correct processes in $p_l.\mathsf{max\_views}(T_4)$ are equal to $v+1$. Since there are at least $2f+1$ correct processes: *(i)* at least $2f+1$ entries in $p_l.\mathsf{max\_views}(T_4)$ are equal to $v+1$, and *(ii)* one of the $f+1$ highest entries in $p_l.\mathsf{max\_views}(T_4)$ is equal to $v+1$. From *(i)*, $p_l.\mathsf{view}^+(T_4) \geq p_l.\mathsf{view}(T_4) \geq v+1$, and from *(ii)*, $p_l.\mathsf{view}(T_4) \leq p_l.\mathsf{view}^+(T_4) \leq v+1$. Hence, $p_l.\mathsf{view}(T_4) = p_l.\mathsf{view}^+(T_4) = v+1$, and therefore, p_l enters $v+1$ by T_4, contradicting (7). □

From Lemmas 9 and 1 we get

Corollary 3 *For any view v, some correct process enters v.*

Finally, the following lemma gives the core argument for the proof of Theorem 2.

Lemma 10 *Consider an execution with an eventual message delay bound δ and a view $\mathcal{V}$ such that*

$$\mathcal{V} \geq \mathsf{GV}(\overline{\mathsf{GST}}) + 1 \wedge F(\mathcal{V}) \geq 2\delta. \qquad (9)$$

Then in this execution FastSync *satisfies Properties 1–5 in Fig. 2 for $d = 2\delta$.*

Proof Property 1 is satisfied trivially. Consider a view $\mathcal{V}$ such that (9) holds. By Corollary 3, some correct process enters $\mathcal{V}$, so that $E_{\text{first}}(\mathcal{V})$ is defined. Since GV is non-decreasing and $\mathcal{V} \geq \mathsf{GV}(\overline{\mathsf{GST}}) + 1$, no correct process can enter $\mathcal{V}$ until after $\overline{\mathsf{GST}}$. Thus, we get that $E_{\text{first}}(\mathcal{V}) \geq \overline{\mathsf{GST}}$.

Fix a view $v \geq \mathcal{V}$. By Corollary 3, some correct process enters v. Then by Lemma 1 and (6) we get

$$\forall v \geq \mathcal{V}.\ E_{\text{first}}(v) \geq E_{\text{first}}(\mathcal{V}) \geq \overline{\mathsf{GST}} \geq \mathsf{GST}. \qquad (10)$$

Then Property 2 holds. Since F is a non-decreasing function, $\forall v \geq \mathcal{V}.\ F(v) \geq F(\mathcal{V}) \geq 2\delta$. Thus, from (10) and Lemma 8, all correct processes enter v and $E_{\text{last}}(v) \leq E_{\text{first}}(v) + 2\delta$. This validates Properties 3 and 4 for $d = 2\delta$.

It remains to prove Property 5. By Corollary 3, some correct process enters view $v+1$. Then by Lemma 3 there exist a time $t < E_{\text{first}}(v+1)$ at which some correct process attempts to advance from v. By (10), $E_{\text{first}}(v) \geq \mathsf{GST}$. Then by Lemma 7 we get $t \geq E_{\text{first}}(v) + F(v)$, which implies $E_{\text{first}}(v+1) > t \geq E_{\text{first}}(v) + F(v)$, as required. We thus conclude that FastSync satisfies Properties 1–5 in Fig. 2 for $d = 2\delta$, as needed. □

Proof of Theorem 2. Consider an execution of FastSync and let δ be the eventual message delay bound in this execution. Then (1) for $u = 2\delta$ implies that $F(v') \geq 2\delta$ for some view v'. Since F is monotone, $\mathcal{V} = \max\{v', \mathsf{GV}(\overline{\mathsf{GST}}) + 1\}$ satisfies (9). Then the required follows from Lemma 10. □

4.2 Proof of FastSync latency bounds

We next extend the proof of FastSync correctness to also establish the latency bounds for entering various views stated by Properties A–C in Fig. 2. Our proofs are structured as follows. Given a view v whose entry time we seek to bound, we first derive a bound on the time by which all correct processes must send a $\texttt{WISH}$ for the view v or higher. We then apply the following lemma, which bounds the latency of entering v as a function of the time by which all correct processes have sent such $\mathtt{WISH}$es.

Lemma 11 *For all views $v > 0$ and times s, if all correct processes p_i send $\mathtt{WISH}(v_i)$ with $v_i \geq v$ no later than at s, and some correct process enters v, then $E_{\text{last}}(v) \leq \max(s, \overline{\mathsf{GST}}) + \delta$.*

Proof Fix an arbitrary correct process p_i that sends $\mathtt{WISH}(v_i)$ with $v_i \geq v$ to all processes at time $t_i \leq s \leq \max(s, \overline{\mathsf{GST}})$. Since $\max(s, \overline{\mathsf{GST}}) \geq \overline{\mathsf{GST}}$, by Lemma 6 there exists a time t_i' such that $\mathsf{GST} \leq t_i' \leq \max(s, \overline{\mathsf{GST}})$ and at t_i', p_i sends $\mathtt{WISH}(v_i')$ with $v_i' \geq v_i \geq v$ to all processes. Since $t_i' \geq \mathsf{GST}$, all correct processes receive $\mathtt{WISH}(v_i')$ from p_i no later than at $t_i' + \delta \leq \max(s, \overline{\mathsf{GST}}) + \delta$.

Consider an arbitrary correct process p_j and let $t_j \leq \max(s, \overline{\mathsf{GST}}) + \delta$ be the earliest time by which p_j receives receives $\mathtt{WISH}(v_i')$ with with $v_i' \geq v$ from each correct processes p_i. Thus, at t_j, the entries of all correct processes in $p_j.\mathsf{max\_views}$ are occupied by views $\geq v$. Since at least $2f + 1$ entries in $p_j.\mathsf{max\_views}$ belong to correct processes, the $(2f + 1)$th highest entry is $\geq v$. Thus, $p_j.\mathsf{view}(t_j) \geq v$. Since $p_j.\mathsf{view}$ is non-decreasing, there exists a time $t_j' \leq t_j$ at which $p_j.\mathsf{view}$ first became $\geq v$. If $p_j.\mathsf{view}(t_j') = p_j.\mathsf{view}^+(t_j') = v$, then p_j enters v at t_j'. Otherwise, either $p_j.\mathsf{view}(t_j') > v$ or $p_j.\mathsf{view}^+(t_j') > v$. Since both $p_j.\mathsf{view}$ and $p_j.\mathsf{view}^+$ are non-decreasing, p_j will never enter v after t_j'. Thus, a correct process cannot enter v after $\max(s, \overline{\mathsf{GST}}) + \delta$. Since by the lemma's premise, some correct process does enter v, $E_{\text{last}}(v) \leq \max(s, \overline{\mathsf{GST}}) + \delta$, as needed. □

The next lemma gives an upper bound on the duration of time a correct process may spend in a view before sending a $\mathtt{WISH}$ for a higher view.

Lemma 12 *If a correct process p_k enters a view $v > 0$, then p_k sends $\mathtt{WISH}(v_k)$ with $v_k \geq v + 1$ no later than at $\max(E_{\text{last}}(v), \mathsf{GST}) + F(v)$.*

Proof Suppose that p_k enters $v > 0$ at time $s_k \leq E_{\text{last}}(v)$, and starts $p_k.\mathsf{timer\_view}$ for the duration of $F(v)$ at s_k. Then

$$p_k.\mathsf{view}(s_k) = p_k.\mathsf{view}^+(s_k) = v.$$

Since the clocks of the correct processes advance at the same rate as real time after GST, $p_k.\mathsf{timer\_view}$ cannot last past $\max(E_{\text{last}}(v), \mathsf{GST}) + F(v)$. Let s_k' such that

$$s_k \leq s_k' \leq \max(E_{\text{last}}(v), \mathsf{GST}) + F(v)$$

be the time at which $p_k.\mathsf{timer\_view}$ either expires or is stopped prematurely by executing the code in line 17. If $p_k.\mathsf{timer\_view}$ expires at s_k', then p_k sends $\mathtt{WISH}(v_k)$ with $v_k = \max(p_k.\mathsf{view}(s_k') + 1, p_k.\mathsf{view}^+(s_k'))$. Since both $p_k.\mathsf{view}$ and $p_k.\mathsf{view}^+$ are non-decreasing, we have $p_k.\mathsf{view}(s_k') \geq v$ and $p_k.\mathsf{view}^+(s_k') \geq v$. Thus, $v_k \geq v + 1$, as required. On the other hand, if $p_k.\mathsf{timer\_view}$ is stopped prematurely at s_k', then $p_k.\mathsf{view}(s_k') > p_k.\mathsf{view}(s_k) = v$ and therefore, $p_k.\mathsf{view}^+(s_k') \geq p_k.\mathsf{view}(s_k') \geq v + 1$. Since $p_k.\mathsf{view}^+$ is non-decreasing and $p_k.\mathsf{view}^+(s_k) = v$, $p_k.\mathsf{view}^+$ must have changed its value from v to $v_k'' \geq v + 1$ at some time s_k'' such that $s_k < s_k'' \leq s_k'$. Thus, the condition in line 20 holds at s_k'', which means that p_k sends $\mathtt{WISH}(v_k)$ with $v_k \geq v + 1$ at s_k''. Thus, in all cases, p_k sends $\mathtt{WISH}(v_k)$ with $v_k \geq v + 1$ no later than at $\max(E_{\text{last}}(v), \mathsf{GST}) + F(v)$, as required. □

We now use the above two lemmas to bound the time it takes for a correct process that has entered a view v to enter the view $v + 1$, as required by Property A.

Corollary 4 *For all times t, if all correct processes enter $v > 0$, $E_{\text{last}}(v) \geq \overline{\mathsf{GST}}$, and some correct process enters $v + 1$, then $E_{\text{last}}(v + 1) \leq E_{\text{last}}(v) + F(v) + \delta$.*

Proof Instantiating Lemma 12 for the special case when all correct processes enter v at $\overline{\mathsf{GST}}$ or later, and given that by (6), $\overline{\mathsf{GST}} \geq \mathsf{GST}$, we get that every correct p_k sends $\mathtt{WISH}(v_k)$ with $v_k \geq v + 1$ no later than at $E_{\text{last}}(v) + F(v) \geq \overline{\mathsf{GST}}$. Then the required follows from Lemma 11. □

We now prove several lemmas needed for Properties B and C. First, given an arbitrary time t such that $t \geq \overline{\mathsf{GST}}$ and $\mathsf{GV}(t) > 0$, we derive an upper bound on the latency of reaching the next view $\mathsf{GV}(t) + 1$ starting from t. To this end, we first bound the time by which all correct processes must send a $\mathtt{WISH}$ for a view $\geq \mathsf{GV}(t) + 1$ (Lemma 13), and then apply Lemma 11 to bound the latency of entering $\mathsf{GV}(t) + 1$ (Corollary 5). In the following, we instantiate this result for $t = \mathsf{GST} + \rho$ to establish Property C for the case of $\mathsf{GV}(\mathsf{GST} + \rho) > 0$ (Corollary 7).

Lemma 13 *Consider a time $t \geq \overline{\mathsf{GST}}$ and suppose that $\mathsf{GV}(t) > 0$. Let $T = t + F(\mathsf{GV}(t)) + \delta$. If some correct process enters $\mathsf{GV}(t) + 1$, then all correct processes p_k send $\mathtt{WISH}(v_k)$ with $v_k \geq \mathsf{GV}(t) + 1$ to all processes no later than at $T + \delta$.*

Proof Since $\mathsf{GV}(t) > 0$, the definition of GV implies that there exists a correct process p_l such that p_l entered $\mathsf{GV}(t)$ and $E_l(\mathsf{GV}(t)) \leq t$. By the view entry condition, $p_l.\mathsf{view}(E_l(\mathsf{GV}(t))) = \mathsf{GV}(t)$, and therefore $p_l.\mathsf{max\_views}$ $(E_l(\mathsf{GV}(t)))$ includes $2f + 1$ entries $\geq \mathsf{GV}(t)$. Since $f + 1$ of these entries belong to correct processes, there exists a set C of $f + 1$ correct processes p_i, each of which sent $\mathtt{WISH}(v_i)$ with $v_i \geq \mathsf{GV}(t)$ to all processes before $E_l(\mathsf{GV}(t)) \leq t$. Since $t \geq \overline{\mathsf{GST}}$, by Lemma 6, p_i sends $\mathtt{WISH}(v_i')$ with

$v'_i \geq v_i \geq \mathsf{GV}(t)$ sometime between GST and t. Since after GST every message sent by a correct process is received by all correct processes within δ of its transmission, the above implies that by $t+\delta$ every correct process receives a $\mathtt{WISH}(v'_i)$ with $v'_i \geq \mathsf{GV}(t)$ from each process $p_i \in C$.

Consider an arbitrary correct process p_j and let $t_j \leq t+\delta$ be the earliest time by which p_j receives $\mathtt{WISH}(v_i)$ with $v_i \geq \mathsf{GV}(t)$ from each process $p_i \in C$. Thus, for all processes $p_i \in C$, $p_j.\mathsf{max\_views}[i](t_j) \geq \mathsf{GV}(t)$. Since $|C| = f+1$, the $(f+1)$th highest entry in $p_j.\mathsf{max\_views}[i](t_j)$ is $\geq \mathsf{GV}(t)$, and therefore, $p_j.\mathsf{view}^+(t_j) \geq \mathsf{GV}(t)$. Then each correct process p_j sends $\mathtt{WISH}(v_j)$ with $v_j \geq \mathsf{GV}(t)$ to all correct processes no later than $t_j \leq t+\delta$. Since $t+\delta > t \geq \overline{\mathsf{GST}}$ and, by the definition of GV, some correct process entered $\mathsf{GV}(t)$, by Lemma 11,

$$E_{\text{last}}(\mathsf{GV}(t)) \leq t + 2\delta. \tag{11}$$

In addition, by Lemma 6, there exists a time t'_j such that $\mathsf{GST} \leq t'_j \leq t+\delta$ and p_j sends $\mathtt{WISH}(v'_j)$ with $v'_j \geq v_j \geq \mathsf{GV}(t)$ at t'_j. Since a message sent by a correct process after GST is received by all correct processes within δ of its transmission, all correct processes must have received $\mathtt{WISH}(v'_j)$ with $v'_j \geq \mathsf{GV}(t)$ from each correct process p_j in-between GST and $t+2\delta$.

Consider an arbitrary correct process p_k. If p_k enters $\mathsf{GV}(t)$, then by Lemma 12, p_k sends $\mathtt{WISH}(v_k)$ with $v_k \geq \mathsf{GV}(t)+1$ at some time $t_k \leq \max(E_{\text{last}}(\mathsf{GV}(t)), \mathsf{GST}) + F(\mathsf{GV}(t))$. If $E_{\text{last}}(\mathsf{GV}(t)) \geq \mathsf{GST}$, then by (11),

$$t_k \leq E_{\text{last}}(\mathsf{GV}(t)) + F(\mathsf{GV}(t)) \leq t + 2\delta + F(\mathsf{GV}(t)) = T + \delta.$$

On the other hand, if $\mathsf{GST} > E_{\text{last}}(\mathsf{GV}(t))$, then since $t \geq \overline{\mathsf{GST}}$, and by (6), $\overline{\mathsf{GST}} \geq \mathsf{GST}$, we have

$$t_k \leq \mathsf{GST} + F(\mathsf{GV}(t)) \leq t + F(\mathsf{GV}(t)) < T + \delta.$$

Thus, we conclude that if p_k enters $\mathsf{GV}(t)$, then the required holds.

Suppose now that p_k never enters $\mathsf{GV}(t)$, and let t_k be the earliest time $\geq$ GST by which p_k receives $\mathtt{WISH}(v'_j)$ from each correct process p_j; we have $t_k \leq t+2\delta$. Since $v'_j \geq \mathsf{GV}(t)$, and there are $2f+1$ correct processes, $p_k.\mathsf{max\_views}(t_k)$ includes at least $2f+1$ entries $\geq \mathsf{GV}(t)$. Thus, $p_k.\mathsf{view}(t_k) \geq \mathsf{GV}(t)$. Since p_k never enters $\mathsf{GV}(t)$, we have either $p_k.\mathsf{view}^+(t_k) \geq p_k.\mathsf{view}(t_k) \geq \mathsf{GV}(t)+1$ or $p_k.\mathsf{view}(t_k) = \mathsf{GV}(t) \wedge p_k.\mathsf{view}^+(t_k) \geq \mathsf{GV}(t)+1$. Thus, $p_k.\mathsf{view}^+(t_k) \geq \mathsf{GV}(t)+1$ and therefore, p_k sends $\mathtt{WISH}(v_k)$ with $v_k \geq \mathsf{GV}(t)+1$ by $t_k \leq t+2\delta \leq T+\delta$. Hence, we get that all correct processes send $\mathtt{WISH}(v_k)$ with $v_k \geq \mathsf{GV}(t)+1$ to all correct processes no later than $T+\delta$, validating the lemma. □

Lemmas 11 and 13 imply an upper bound on the latency of reaching the view $\mathsf{GV}(t)+1$ from an arbitrary time t such that $t \geq \overline{\mathsf{GST}}$ and $\mathsf{GV}(t) > 0$.

Corollary 5 *Consider a time $t \geq \overline{\mathsf{GST}}$ and suppose that $\mathsf{GV}(t) > 0$. If some correct process enters the view $\mathsf{GV}(t)+1$, then $E_{\text{last}}(\mathsf{GV}(t)+1) \leq t + F(\mathsf{GV}(t)) + 3\delta$.*

We next derive a bound on the latency of entering view 1. This is used to prove Property B as well as Property C for the case of $\mathsf{GV}(\mathsf{GST}+\rho) = 0$.

Lemma 14 *If some correct process enters view 1, then $E_{\text{last}}(1) \leq \min(t_1+2\delta, t_2+\delta)$, where $t_1 = \max(S_{f+1}, \overline{\mathsf{GST}})$ and $t_2 = \max(S_{\text{last}}, \overline{\mathsf{GST}})$. Equivalently:*

$$E_{\text{last}}(1) \leq t_2 + \delta; \tag{12}$$

$$E_{\text{last}}(1) \leq t_1 + 2\delta. \tag{13}$$

Proof We consider three cases:

- $\overline{\mathsf{GST}} \leq S_{f+1} \leq S_{\text{last}}$, so that $t_1 = S_{f+1}$ and $t_2 = S_{\text{last}}$. We consider two cases:
  - $S_{f+1} + \delta < S_{\text{last}}$. Let C be the set of the $f+1$ correct processes p_i calling $\mathtt{start}()$ at $t_i \leq S_{f+1}$. If $p_i.\mathsf{view}^+(t_i) = 0$, then at t_i, p_i sends $\mathtt{WISH}(1)$ to all processes by executing the code in line 3. Otherwise, $p_i.\mathsf{view}^+(t_i) \geq 1$, and p_i sent $\mathtt{WISH}(v_i)$ with $v_i \geq 1$ when $p_i.\mathsf{view}^+$ first became equal to v_i at some time $s_i < t_i \leq S_{f+1}$. Since $t_i \leq S_{f+1}$ and $S_{f+1} \geq \overline{\mathsf{GST}}$, in both cases, by Lemma 6, p_i sends $\mathtt{WISH}(v'_i)$ with $v'_i \geq v_i \geq 1$ sometime between GST and S_{f+1}. Thus, we get that all processes $p_i \in C$ send $\mathtt{WISH}(v'_i)$ with $v'_i \geq 1$ to all processes in-between GST and S_{f+1}. It follows that all correct processes receive all these $\mathtt{WISH}(v'_i)$ messages no later than $S_{f+1}+\delta$. Consider a correct process p_j, and let t_j be the earliest time by which p_j receives the $\mathtt{WISH}(v'_i)$ messages sent by the processes $p_i \in C$ in-between GST and S_{f+1}; then $\mathsf{GST} \leq t_j \leq S_{f+1}+\delta$. Thus, $p_j.\mathsf{max\_views}[k](t_j) \geq 1$ for all $p_k \in C$. Since $|C| \geq f+1$, the $(f+1)$th highest entry in $p_j.\mathsf{max\_views}[k](t_j)$ is ≥ 1, and therefore, $p_j.\mathsf{view}^+(t_j) \geq 1$. Thus, p_j sends $\mathtt{WISH}(v_j)$ with $v_j \geq 1$ to all processes no later than $t_j \leq S_{f+1}+\delta$, and we also have $S_{f+1}+\delta > \overline{\mathsf{GST}}$. Since some correct process enters view 1, by Lemma 11, $E_{\text{last}}(1) \leq S_{f+1}+2\delta = t_1+2\delta$. Since $t_1+\delta < t_2$, we also have $E_{\text{last}}(1) \leq \min(t_1+2\delta, t_2+\delta)$, as needed.
  - $S_{f+1}+\delta \geq S_{\text{last}}$. Let p_i be a correct process calling $\mathtt{start}()$ at $t_i \leq S_{\text{last}}$. If $p_i.\mathsf{view}^+(t_i) = 0$, then at

t_i, p_i sends $\mathtt{WISH}(1)$ to all processes by executing the code in line 3. Otherwise, $p_i.\mathsf{view}^+(t_i) \geq 1$, and p_i sent $\mathtt{WISH}(v_i)$ with $v_i \geq 1$ when $p_i.\mathsf{view}^+$ first became equal to v_i at some time $s_i < t_i \leq S_{\text{last}}$. Thus, all correct processes send $\mathtt{WISH}(v_i)$ with $v_i \geq 1$ to all processes no later than $S_{\text{last}} \geq \overline{\mathsf{GST}}$. Since some correct process enters view 1, by Lemma 11, $E_{\text{last}}(1) \leq S_{\text{last}} + \delta = t_2 + \delta$. Since $t_1 + \delta \geq t_2$, we also have $E_{\text{last}}(1) \leq \min(t_1 + 2\delta, t_2 + \delta)$, as needed.

- $S_{f+1} < \overline{\mathsf{GST}} \leq S_{\text{last}}$, so that $t_1 = \overline{\mathsf{GST}}$ and $t_2 = S_{\text{last}}$. We consider two cases:
  - $\overline{\mathsf{GST}} + \delta < S_{\text{last}}$. Let C be the set of the $f+1$ correct processes p_i calling $\mathtt{start}()$ at $t_i < \overline{\mathsf{GST}}$. If $p_i.\mathsf{view}^+(t_i) = 0$, then at t_i, p_i sends $\mathtt{WISH}(1)$ to all processes by executing the code in line 3. Otherwise, $p_i.\mathsf{view}^+(t_i) \geq 1$, and p_i sent $\mathtt{WISH}(v_i)$ with $v_i \geq 1$ when $p_i.\mathsf{view}^+$ first became equal to v_i at sometime before t_i. Since $t_i < \overline{\mathsf{GST}}$, by Lemma 6, there exists a time s_i such that $\mathsf{GST} \leq s_i < \overline{\mathsf{GST}}$ and at s_i, p_i sends $\mathtt{WISH}(v_i')$ with $v_i' \geq v_i \geq 1$ to all processes. Thus, we get that all processes in C send $\mathtt{WISH}(v_i')$ with $v_i \geq 1$ to all processes in-between GST and $\overline{\mathsf{GST}}$. It follows that all correct processes receive all these $\mathtt{WISH}(v_i')$ messages no later than $\overline{\mathsf{GST}} + \delta$. Consider a correct process p_j, and let t_j be the earliest time by which p_j receives the $\mathtt{WISH}(v_i')$ messages sent by the processes in C in-between GST and $\overline{\mathsf{GST}}$; then $\mathsf{GST} \leq t_j \leq \overline{\mathsf{GST}} + \delta$. Thus, $p_j.\mathsf{max\_views}[k](t_j) \geq 1$ for all $p_k \in C$. Since $|C| \geq f+1$, the $(f+1)$th highest entry in $p_j.\mathsf{max\_views}[k](t_j)$ is ≥ 1. Thus, $p_j.\mathsf{view}^+(t_j) \geq 1$, so that p_j sends $\mathtt{WISH}(v_j)$ with $v_j \geq 1$ to all processes no later than $t_j \leq \overline{\mathsf{GST}} + \delta$. Since some correct process enters view 1, by Lemma 11, $E_{\text{last}}(1) \leq \overline{\mathsf{GST}} + 2\delta = t_1 + 2\delta$. Since $t_1 + \delta < t_2$, we also have $E_{\text{last}}(1) \leq \min(t_1 + 2\delta, t_2 + \delta)$, as needed.
  - $\overline{\mathsf{GST}} + \delta \geq S_{\text{last}}$. Let p_i be a correct process calling $\mathtt{start}()$ at $t_i \leq S_{\text{last}}$. If $p_i.\mathsf{view}^+(t_i) = 0$, then at t_i, p_i sends $\mathtt{WISH}(1)$ to all processes by executing the code in line 3. Otherwise, $p_i.\mathsf{view}^+(t_i) \geq 1$, and p_i sent $\mathtt{WISH}(v_i)$ with $v_i \geq 1$ when $p_i.\mathsf{view}^+$ first became equal to v_i at some time $s_i < t_i \leq S_{\text{last}}$. Thus, we get that all correct processes p_i send $\mathtt{WISH}(v_i)$ with $v_i \geq 1$ to all processes no later than $S_{\text{last}} \geq \overline{\mathsf{GST}}$. Since some correct process enters view 1, by Lemma 11, $E_{\text{last}}(1) \leq S_{\text{last}} + \delta = t_2 + \delta$. Since $t_2 \leq t_1 + \delta$, we also have $E_{\text{last}}(1) \leq \min(t_1 + 2\delta, t_2 + \delta)$, as needed.
- $S_{f+1} \leq S_{\text{last}} < \overline{\mathsf{GST}}$, so that $t_1 = t_2 = \overline{\mathsf{GST}}$. Let p_i be a correct process calling $\mathtt{start}()$ at $t_i \leq S_{\text{last}} < \overline{\mathsf{GST}}$. If $p_i.\mathsf{view}^+(t_i) = 0$, then at t_i, p_i sends $\mathtt{WISH}(1)$ to all processes by executing the code in line 3. Otherwise, $p_i.\mathsf{view}^+(t_i) \geq 1$, and p_i sent $\mathtt{WISH}(v_i)$ with $v_i \geq 1$ when $p_i.\mathsf{view}^+$ first became equal to v_i at some time $s_i < t_i \leq S_{\text{last}} < \overline{\mathsf{GST}}$. By Lemma 6, in both cases p_i sends $\mathtt{WISH}(v_i')$ with $v_i' \geq v_i \geq 1$ sometime between GST and $\overline{\mathsf{GST}}$. Since some correct process enters view 1, by Lemma 11, we get that $E_{\text{last}}(1) \leq \overline{\mathsf{GST}} + \delta = t_2 + \delta$. Since $t_1 = t_2$, we also have $E_{\text{last}}(1) \leq \min(t_1 + 2\delta, t_2 + \delta)$, as needed.

Thus, we get that in all three cases above, $E_{\text{last}}(1) \leq \min(t_1 + 2\delta, t_2 + \delta)$, as needed. □

We now instantiate the bound (12) of Lemma 14 for the special case of all correct processes starting the protocol after GST to obtain the latency bound stipulated by Property B.

Corollary 6 *If $S_{\text{first}} \geq \mathsf{GST}$ and a correct process enters view 1, then $E_{\text{last}}(1) \leq S_{\text{last}} + \delta$.*

Proof Let $S_{\text{first}} \geq \mathsf{GST}$. Then $\overline{\mathsf{GST}} = S_{\text{first}}$ by the definition of $\overline{\mathsf{GST}}$. We now use the bound (12) of Lemma 14 with $t_1 = \max(S_{f+1}, S_{\text{first}}) = S_{f+1}$ and $t_2 = \max(S_{\text{last}}, S_{\text{first}}) = S_{\text{last}}$. We get $E_{\text{last}}(1) \leq t_2 + \delta = S_{\text{last}} + \delta$, as required. □

Finally, by combining Corollary 5 and the bound (13) of Lemma 14, we obtain the following bound, used to prove Property C.

Corollary 7 *Assume that $S_{\text{first}} < \mathsf{GST}$ and $S_{f+1} \leq \mathsf{GST} + \rho$. Then if a correct process enters $\mathsf{GV}(\mathsf{GST} + \rho) + 1$, then*

$$E_{\text{last}}(\mathsf{GV}(\mathsf{GST} + \rho) + 1) \leq \mathsf{GST} + \rho + F(\mathsf{GV}(\mathsf{GST} + \rho)) + 3\delta.$$

Proof Since $S_{\text{first}} < \mathsf{GST}$, by the definition of $\overline{\mathsf{GST}}$, we have $\overline{\mathsf{GST}} = \mathsf{GST} + \rho$. If $\mathsf{GV}(\mathsf{GST} + \rho) = 0$, then choosing $t_1 = \max(S_{f+1}, \mathsf{GST} + \rho)$, and $t_2 = \max(S_{\text{last}}, \mathsf{GST} + \rho)$, by the bound (13) of Lemma 14 we get $E_{\text{last}}(\mathsf{GV}(\mathsf{GST} + \rho) + 1) \leq t_1 + 2\delta$. Since $S_{f+1} \leq \mathsf{GST} + \rho$, we have $t_1 = \mathsf{GST} + \rho$, and therefore,

$$E_{\text{last}}(\mathsf{GV}(\mathsf{GST} + \rho) + 1) \leq \mathsf{GST} + \rho + 2\delta <$$
$$\mathsf{GST} + \rho + F(\mathsf{GV}(t)) + 3\delta,$$

as required. On the other hand, if $\mathsf{GV}(\mathsf{GST} + \rho) > 0$, then by Corollary 5,

$$E_{\text{last}}(\mathsf{GV}(\mathsf{GST} + \rho) + 1) \leq \mathsf{GST} + \rho + F(\mathsf{GV}(\mathsf{GST} + \rho)) + 3\delta,$$

as required. □

Proof of Theorem 1. Consider an execution of FastSync and let δ be the eventual message delay bound in this execution. We first show how to select a view $\mathcal{V}$ such that (9) holds. We consider the following cases. First, if

$$S_{\text{first}} \geq \mathsf{GST} \wedge F(1) \geq 2\delta, \tag{14}$$

then we let $\mathcal{V} = 1$. Since $S_{\text{first}} \geq \mathsf{GST}$, the definition of $\overline{\mathsf{GST}}$ implies $\overline{\mathsf{GST}} = S_{\text{first}}$, and therefore, (9) holds. Second, if

$$S_{\text{first}} < \mathsf{GST} \wedge S_{f+1} \leq \mathsf{GST} + \rho \wedge$$
$$F(\mathsf{GV}(\mathsf{GST} + \rho) + 1) \geq 2\delta, \tag{15}$$

then we let $\mathcal{V} = \mathsf{GV}(\mathsf{GST} + \rho) + 1$. Since $S_{\text{first}} < \mathsf{GST}$, the definition of $\overline{\mathsf{GST}}$ implies $\overline{\mathsf{GST}} = \mathsf{GST} + \rho$, and therefore, (9) holds. In all other cases, (1) for $u = 2\delta$ implies that $F(v') \geq 2\delta$ for some view v', and therefore, by the monotonicity of F, $\mathcal{V} = \max\{v', \mathsf{GV}(\overline{\mathsf{GST}}) + 1\}$ satisfies (9).

Thus, by Lemma 10, Properties 1–5 in Fig. 2 hold for $\mathcal{V}$ chosen as above and $d = 2\delta$.

To prove Property A, fix $v \geq \mathcal{V}$. By Property 3, all correct processes enter v. By (9), $v \geq \mathcal{V} \geq \mathsf{GV}(\overline{\mathsf{GST}}) + 1$. Given that GV is non-decreasing, this implies that no correct process can enter v until after $\overline{\mathsf{GST}}$. Thus, $E_{\text{last}}(v) \geq E_{\text{first}}(v) > \overline{\mathsf{GST}}$, and by Corollary 4 we get $E_{\text{last}}(v+1) \leq E_{\text{last}}(v) + F(v) + \delta$, validating Property A. Finally, by our choice of $\mathcal{V}$, (14) implies $\mathcal{V} = 1$, and (15) implies $\mathcal{V} = \mathsf{GV}(\mathsf{GST} + \rho) + 1$. Thus, Property B follows from Corollary 6, and Property C from Corollary 7. We therefore, conclude that FastSync satisfies all properties in Fig. 2 for $d = 2\delta$ and $\mathcal{V}$ chosen as above. □

5 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 the Byzantine consensus problem that relies on an application-specific valid() predicate to indicate whether a value is valid [16,23]. 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.

5.1 Single-shot PBFT

We first consider the seminal Practical Byzantine Fault Tolerance (PBFT) protocol [19]. This protocol was originally presented as implementing state-machine replication, where the processes agree on a sequence of commands. In Algorithm 2 we present its specialization to consensus, in the style of DLS [27]. The protocol in Algorithm 2 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 some process decided a value in a previous view, the leader will propose the same value. To enable the leader to do this, when a process receives a notification from the synchronizer to move to a view v (line 1), it sends a NEWLEADER message to the leader of v with some information about the latest value it accepted in a previous view. 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. Various messages sent in the protocol are tagged with the view of the sender; a receiver accepts a message only if it is in the same view according to the curr_view variable.

The leader computes its proposal 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). We describe the proposal computation in detail after describing the rest of the protocol. The leader's proposal is processed in two phases, each with an all-to-all message exchange among processes. 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 (also 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 following predicate:

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

Once a process assembles a prepared certificate, the process stores it in cert, the view in which it was formed

```
1  upon new_view($v$)
2      curr_view ← $v$;
3      voted ← FALSE;
4      send ⟨NEWLEADER(curr_view, prepared_view,
         prepared_val, cert)⟩$_i$ to leader(curr_view);
5  when received $\{\langle \texttt{NEWLEADER}(v, view_j, val_j, cert_j)\rangle_j \mid p_j \in Q\} = M$ for a quorum $Q$
6      pre: $\text{curr\_view} = v \wedge p_i = \text{leader}(v) \wedge (\forall m \in M.\, \text{ValidNewLeader}(m))$;
7      if $\exists j.\, view_j = \max\{view_k \mid p_k \in Q\} \neq 0$ then
8          send $\langle \texttt{PROPOSE}(v, val_j, M)\rangle_i$ to all;
9      else
10         send $\langle \texttt{PROPOSE}(v, \texttt{myval}(), M)\rangle_i$ to all;
11 when received $\langle \texttt{PROPOSE}(v, x, \_)\rangle_j = m$
12     pre: $\text{curr\_view} = v \wedge \text{voted} = \text{FALSE} \wedge \text{SafeProposal}(m)$;
13     curr_val ← $x$;
14     voted ← TRUE;
15     send $\langle \texttt{PREPARED}(v, \text{hash}(\text{curr\_val}))\rangle_i$ to all;
16 when received $\{\langle \texttt{PREPARED}(v, h)\rangle_j \mid p_j \in Q\} = C$ for a quorum $Q$
17     pre: $\text{curr\_view} = v \wedge \text{voted} = \text{TRUE} \wedge \text{hash}(\text{curr\_val}) = h$;
18     prepared_val ← curr_val;
19     prepared_view ← curr_view;
20     cert ← $C$;
21     send $\langle \texttt{COMMITTED}(v, h)\rangle_i$ to all;
22 when received $\{\langle \texttt{COMMITTED}(v, h)\rangle_j \mid p_j \in Q\}$ for a quorum $Q$
23     pre: $\text{curr\_view} = \text{prepared\_view} = v \wedge \text{hash}(\text{curr\_val}) = h$;
24     decide(curr_val);
```

Algorithm 2: Single-shot PBFT at a process p_i. All variables storing views are initially set to 0 and others to $\perp$. The predicates ValidNewLeader and SafeProposal are defined by (16) and (17).

in prepared_view, and the value it corresponds to in prepared_val. At this point we say that the process *prepared* the value. Since a prepared certificate consists of at least $2f+1$ PREPARED messages and there are $3f+1$ processes 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 hashes in the same view, which cannot happen due to the check on the voted flag in line 12. Formally, let us write $\text{wf}(C)$ (for *well-formed*) if the set of correctly signed messages C was generated in the execution of the protocol.

Proposition 4

$$\forall v, C, C', x, x'.\, \text{prepared}(C, v, \text{hash}(x)) \wedge \text{prepared}(C', v, \text{hash}(x')) \wedge \text{wf}(C) \wedge \text{wf}(C') \implies x = x'.$$

Since the synchronizer transitions processes through increasing views (Property 1 in Fig. 2), we also get

Proposition 5 *The variables* curr_view *and* prepared_view *at a correct process never decrease and we always have* prepared_view $\leq$ curr_view.

Having prepared a value, the process participates in another 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 this value (line 22). We call such a quorum a *committed certificate* and let

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

Once a process assembles a committed certificate for a value x, it decides on x.

When the synchronizer triggers a new_view notification at a process (line 1), the process sends a NEWLEADER message to the new leader. Since preparing a value is a prerequisite for deciding on it, by Proposition 4 forming a prepared certificate for a value x in a view v guarantees that x is the only value that can possibly be 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 well-formed NEWLEADER messages (line 5), as checked by a ValidNewLeader predicate:

$$\text{ValidNewLeader}(\langle \texttt{NEWLEADER}(v', v, x, C)\rangle\_) \iff v < v' \wedge (v \neq 0 \implies \text{prepared}(C, v, \text{hash}(x))). \quad (16)$$

Similarly to Paxos [37], 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(). Since a faulty leader may not follow this rule, the processes need to check that the leader has selected the proposal correctly. To this end, the leader's PROPOSE message carries, in addition to the proposal, the NEWLEADER messages on the basis of which it was computed (line 4). Processes check its correctness by redoing the leader's computation, as specified by the SafeProposal predicate (line 12):

$$\begin{aligned}&\mathsf{SafeProposal}(\langle \texttt{PROPOSE}(v, x, M)\rangle_j) \iff\\ &\quad p_j = \mathsf{leader}(v) \wedge \mathsf{valid}(x) \wedge\\ &\quad \exists Q, view, val, cert.\, \mathsf{quorum}(Q) \wedge\\ &\quad M = \{\langle \texttt{NEWLEADER}(v, view_k, val_k, cert_k)\rangle_k \mid p_k \in Q\} \wedge\\ &\quad (\forall m \in M.\, \mathsf{ValidNewLeader}(m)) \wedge\\ &\quad ((\exists k.\, view_k \neq 0) \implies\\ &\qquad (\exists k.\, view_k = \max\{view_l \mid p_l \in Q\} \wedge x = val_k)). \qquad (17)\end{aligned}$$

Note that the SafeProposal check still allows a faulty leader to send different proposals to different processes, e.g., by selecting different quorums of NEWLEADER messages, or by selecting different values in case when none of the NEWLEADER messages carries a prepared certificate. Hence, the SafeProposal check does not make the exchange of PREPARED messages unnecessary, as it protects against such behavior. On the other hand, exchanging COMMITTED messages before deciding is needed to preserve decisions across views and thereby validate the Agreement property of consensus. It guarantees that, if a process decides on a value x, then at least $f + 1$ correct processes have prepared x. Regardless of whether the leader of the next view is correct or faulty, due to the SafeProposal check it will make a proposal based on prepared values from at least $2f + 1$ processes. Then at least one correct process out of these will report the value x, which the leader will have to take into account. This is the core argument in the proof of Agreement we give below (Theorem 9).

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. Processes can avoid sending NEWLEADER messages to this leader, and can accept its proposal after checking that it satisfies valid rather than ValidNewLeader. 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 15), this does not violate liveness. Thus, assuming consensus proposals of bounded size, the protocol Algorithm 2 runs in bounded space, and so does the overall consensus protocol with the FASTSYNC synchronizer.

Safety In PBFT, deciding on a value requires preparing it, which implies

Proposition 6

$$\forall v, C, h.\, \mathsf{committed}(C, v, h) \wedge \mathsf{wf}(C) \implies \exists C'.\, \mathsf{prepared}(C', v, h) \wedge \mathsf{wf}(C').$$

Furthermore, the validity check in SafeProposal ensures that any prepared value is valid:

Proposition 7

$$\forall v, C, x.\, \mathsf{prepared}(C, v, \mathsf{hash}(x)) \wedge \mathsf{wf}(C) \implies \mathsf{valid}(x).$$

The above two propositions imply

Corollary 8 *Single-shot PBFT satisfies Validity.*

The Agreement property of consensus follows from the next lemma, ensuring that if a committed certificate was assembled for a value in a given view, then only this value can be prepared in any higher view.

Lemma 15

$$\begin{aligned}\forall v, v', C, C', x, x'.\, &\mathsf{committed}(C, v, \mathsf{hash}(x)) \wedge\\ &\mathsf{prepared}(C', v', \mathsf{hash}(x')) \wedge\\ &\mathsf{wf}(C) \wedge \mathsf{wf}(C') \wedge v < v' \implies x = x'.\end{aligned}$$

Proof Fix v, C and x, and assume $\mathsf{committed}(C, v, \mathsf{hash}(x))$ and $\mathsf{wf}(C)$; then $v > 0$. We prove by induction on v' that

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

Assume this holds for all $v' < v^*$; we now prove it for $v' = v^*$. To this end, assume $v < v'$, $\mathsf{prepared}(C', v', \mathsf{hash}(x'))$ and $\mathsf{wf}(C')$.

From the induction hypothesis it follows that

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

Furthermore, by Propositions 4 and 6 we have

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

so that overall we get

$$\forall C'', v'', x''.\, v \le v'' < v' \wedge \mathsf{prepared}(C'', v'', \mathsf{hash}(x'')) \wedge \mathsf{wf}(C'') \implies x = x''. \qquad (18)$$

Since $\mathsf{prepared}(C', v', \mathsf{hash}(x'))$ and $\mathsf{wf}(C')$, some correct process has received a message $m = \langle \texttt{PROPOSE}(v', x', M)\rangle\_$ from the leader of v' such that $\mathsf{SafeProposal}(m)$ (line 12). Let

$$M = \{\langle \mathtt{NEWLEADER}(v', view_j, val_j, cert_j)\rangle_j \mid p_j \in Q'\}$$

for some quorum Q'. Since $\mathsf{SafeProposal}(m)$, we have $\forall m' \in M.\, \mathsf{ValidNewLeader}(m')$, so that

$$\forall p_j \in Q'.\, view_j < v' \wedge (view_j \neq 0 \Longrightarrow \mathsf{prepared}(cert_j, view_j, \mathsf{hash}(val_j)) \wedge \mathsf{wf}(cert_j)).$$

From this and (18) we get that

$$\forall p_j \in Q'.\, view_j \geq v \Longrightarrow val_j = x. \tag{19}$$

Since $\mathsf{committed}(C, v, \mathsf{hash}(x))$ and $\mathsf{wf}(C)$, a quorum Q of processes sent $\mathtt{COMMITTED}(v, \mathsf{hash}(x))$. The quorums Q and Q' have to intersect in some correct process p_k, which has thus sent both $\mathtt{COMMITTED}(v, \mathsf{hash}(x))$ and $\mathtt{NEWLEADER}(v', view_k, val_k, cert_k)$. Since $v < v'$, this process p_k must have sent the $\mathtt{COMMITTED}$ message before the $\mathtt{NEWLEADER}$ message. Before sending $\mathtt{COMMITTED}(v, \mathsf{hash}(x))$ the process set $\mathsf{prepared\_view}$ to v (line 19). Then by Proposition 5 process p_k must have had $\mathsf{prepared\_view} \geq v$ when it sent the $\mathtt{NEWLEADER}$ message. Hence, $view_k \geq v \neq 0$ and $\max\{view_l \mid p_l \in Q'\} \geq v$. Then from (19) for any j such that $view_j = \max\{view_l \mid p_l \in Q'\}$ we must have $val_j = x$. Since $\mathsf{SafeProposal}(m)$ holds, this implies $x' = x$. □

Corollary 9 *Single-shot PBFT satisfies Agreement.*

Proof Assume two correct processes decide on values x and x' in views v and v', respectively. Then

$$\mathsf{committed}(C, v, \mathsf{hash}(x)) \wedge \mathsf{committed}(C', v', \mathsf{hash}(x'))$$

for some well-formed C and C'. By Proposition 6 we have

$$\mathsf{prepared}(C_0, v, \mathsf{hash}(x)) \wedge \mathsf{prepared}(C_0', v', \mathsf{hash}(x'))$$

for some well-formed C_0 and C_0'. Without loss of generality assume $v \leq v'$. If $v = v'$, then $x = x'$ by Proposition 4. If $v < v'$, then $x = x'$ by Lemma 15. □

Liveness and latency Assume that the protocol is used with a synchronizer satisfying Properties 1–5 in Fig. 2. To simplify the following latency analysis, we assume $d = 2\delta$, as for FastSync (the liveness proof in the general case is virtually identical to the one given here). The next theorem states requirements on a view sufficient for the protocol to reach a decision and quantifies the resulting latency.

Theorem 3 *Consider an execution of single-shot PBFT with an eventual message delay bound δ, and let $v \geq \mathcal{V}$ be a view such that $F(v) \geq 6\delta$ and $\mathsf{leader}(v)$ is correct. Then all correct processes decide in view v by $E_{\mathrm{last}}(v) + 4\delta$.*

Proof By Property 2, we have $E_{\mathrm{first}}(v) \geq \mathsf{GST}$, so that all messages sent by correct processes after $E_{\mathrm{first}}(v)$ get delivered to all correct processes within δ. Once a correct process enters v, it sends its $\mathtt{NEWLEADER}$ message, so that $\mathsf{leader}(v)$ is guaranteed to receive a quorum of such messages by $E_{\mathrm{last}}(v) + \delta$. When this happens, the leader will send its proposal in a $\mathtt{PROPOSE}$ message, which correct processes will receive by $E_{\mathrm{last}}(v) + 2\delta$. If they deem the proposal safe, it takes them at most 2δ to exchange the sequence of $\mathtt{PREPARED}$ and $\mathtt{COMMITTED}$ messages leading to decisions. By (2), all correct processes will stay in v until at least $E_{\mathrm{last}}(v) + (F(v) - d) \geq E_{\mathrm{last}}(v) + 4\delta$ and thus will not send a message with a view $> v$ until this time. Then 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 $E_{\mathrm{last}}(v) + 4\delta$.

It remains to show that the proposal x that $\mathsf{leader}(v)$ makes in view v (line 5) will be deemed safe by all correct processes according to the $\mathsf{SafeProposal}$ predicate (line 11). All the conjuncts of $\mathsf{SafeProposal}$ except for $\mathsf{valid}(x)$ are trivially satisfied given that $\mathsf{leader}(v)$ is correct. If the leader is choosing its own proposal as x, then it is valid because correct processes propose valid values. Otherwise, from $\mathsf{ValidNewLeader}$ we get that $\mathsf{prepared}(C, \_, \mathsf{hash}(x))$ for a well-formed C. Hence, by Proposition 7 we again have $\mathsf{valid}(x)$. □

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 3 implies that the protocol is live when the timeouts are high enough.

Corollary 10 *Consider an execution with an eventual message delay bound δ, and assume that (1) holds for $u = 6\delta$. Then in this execution single-shot PBFT satisfies Termination.*

We can ensure that single-shot PBFT is always live by picking a function F that grows without bound. Alternatively, we can prevent timeouts from growing unboundedly by picking F that satisfies (3) for $U = 6\Delta$: then by Proposition 1, (1) holds for $u = 6\delta$.

When single-shot PBFT is used with the FastSync synchronizer, rather than an arbitrary one, we can use Properties A–C in Fig. 2 to bound how quickly the protocol reaches a decision after GST. To this end, we combine Theorem 3 with Property C, which bounds the latency of view synchronization when the protocol is started before GST, and Proposition 2, which bounds the latency of going through up to f views with faulty leaders.

Corollary 11 *Let $v = \mathsf{GV}(\mathsf{GST} + \rho) + 1$ and assume that $S_{\mathrm{first}} < \mathsf{GST}$, $F(v) \geq 6\delta$ and $S_{f+1} \leq \mathsf{GST} + \rho$. Then in* *single-shot PBFT all correct processes decide by* $\mathsf{GST} + \rho + \sum_{k=v-1}^{v+f-1}(F(k)+\delta) + 6\delta$.

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 3 exploits this property to bound the latency of PBFT 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 12 *Assume that* $S_{\text{first}} \geq \mathsf{GST}$ *and* $F(1) \geq 6\delta$. *Then in single-shot PBFT all correct processes decide by* $S_{\text{last}} + \sum_{k=1}^{f}(F(k)+\delta) + 5\delta$.

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 3 for the special case of $v = \mathcal{V} = 1$.

Corollary 13 *Assume that* $S_{\text{first}} \geq \mathsf{GST}$, $F(1) \geq 5\delta$ *and* $\mathsf{leader}(1)$ *is correct. Then in single-shot PBFT all correct processes decide by* $S_{\text{last}} + 4\delta$.

Assume that F is such that (3) holds for $U = 6\Delta$; then as we noted above, single-shot PBFT with FastSync is live. Furthermore, under the conditions of Corollaries 11 and 12, the latency of the protocol is bounded even when it has to go through multiple views before deciding: by $\mathsf{GST} + \rho + (6\Delta+\delta)(f+1) + 6\delta$ if starting before GST, and by $S_{\text{last}} + (6\Delta+\delta)f + 5\delta$ if starting after GST.

Communication complexity To reach a decision in a single view with a correct leader, the protocol requires exchanging messages with $O(n^3)$ signatures: this is because the leader has to forward prepared certificates to processes together with its proposal to prove the correctness of the latter. This can be lowered to $O(n^2)$ by using threshold signatures, which combine multiple signatures into one [31]. We next consider a protocol that lowers the communication complexity further.

5.2 Single-shot HotStuff

We consider the more recent HotStuff protocol [45]. The protocol was originally presented as solving an inherently multi-shot problem, agreeing on a hash-chain of blocks. In Algorithm 3 we present its single-shot version that concisely expresses the key idea and allows comparing the protocol with others. For brevity, we also eschew the use of threshold signatures, used in the original presentation. This still yields a protocol with lower communication complexity than PBFT.

HotStuff delegated view synchronization to a separate component [45], 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 Sect. 3 and give precise bounds on its latency. We also show that the protocol requires only bounded space when using our synchronizer FastSync.

Like single-shot PBFT, the protocol in Algorithm 3 switches views when told by the synchronizer, with view leaders rotating round-robin. However, the leader's proposal is processed in three phases instead of two. The protocol starts in the same way as PBFT: processes send `NEWLEADER` messages with prepared certificates to the leader (line 4), the leader chooses its proposal as before and distributes it in a `PROPOSE` message (line 11). The processes then assemble prepare certificates by exchanging `PREPARED` messages (line 16), so that Proposition 4 still holds. However, after forming a prepared certificate for a value, a process participates in an additional 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 $\mathsf{locked\_view}$ 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 $\mathsf{SafeProposal}$ check the process does on a `PROPOSE` message from a leader (line 12):

$$\begin{aligned}&\mathsf{SafeProposal}(\langle \mathtt{PROPOSE}(v,x,C)\rangle_j) \iff\\ &\quad p_j = \mathsf{leader}(v) \wedge \mathsf{valid}(x) \wedge\\ &\quad (\mathsf{locked\_view} \neq 0 \Longrightarrow x = \mathsf{prepared\_val} \vee\\ &\quad (\exists v'.\, v > v' > \mathsf{locked\_view} \wedge \mathsf{prepared}(C, v', \mathsf{hash}(x)))).\end{aligned} \tag{20}$$

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.

```
upon new_view($v$)
    curr_view ← $v$;
    voted ← FALSE;
    send ⟨NEWLEADER(curr_view, prepared_view,
      prepared_val, cert)⟩$_i$ to leader(curr_view);

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

when received ⟨PROPOSE($v, x, \_$)⟩$_j = m$
    pre: curr_view $= v \wedge$ voted $=$ FALSE $\wedge$
         SafeProposal($m$);
    curr_val ← $x$;
    voted ← TRUE;
    send ⟨PREPARED($v$, hash(curr_val))⟩$_i$ to all;

when received $\{\langle$PREPARED$(v, h)\rangle_j \mid p_j \in Q\} = C$ for a quorum $Q$
    pre: curr_view $= v \wedge$ voted $=$ TRUE $\wedge$
         hash(curr_val) $= h$;
    prepared_val ← curr_val;
    prepared_view ← curr_view;
    cert ← $C$;
    send ⟨PRECOMMITTED($v, h$)⟩$_i$ to all;

when received $\{\langle$PRECOMMITTED$(v, h)\rangle_j \mid p_j \in Q\}$ for a quorum $Q$
    pre: curr_view = prepared_view $= v \wedge$
         hash(curr_val) $= h$;
    locked_view ← prepared_view;
    send ⟨COMMITTED($v, h$)⟩$_i$ to all;

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

Algorithm 3: Single-shot HotStuff at a process p_i. All variables storing views are initially set to 0 and others to $\perp$. The predicates ValidNewLeader and SafeProposal are defined by (16) and (20).

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 assembles a committed certificate (line 26). Once a process assembles a committed certificate for a value x, it decides on x. Assembling a committed certificate for 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).

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 4), this does not violate liveness. Thus, assuming consensus proposals of bounded size, the protocol in Algorithm 3 runs in bounded space, and so does the overall consensus protocol with the FastSync synchronizer.

Safety First note that, due to Property 1 of the synchronizer, we have

Proposition 8 *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.

The Validity property of consensus is proved like for PBFT (Sect. 5.1). The Agreement property follows from the following analog of Lemma 15.

Lemma 16

$$\forall v, v', C, C', x, x'.\ \mathsf{committed}(C, v, \mathsf{hash}(x)) \wedge \mathsf{prepared}(C', v', \mathsf{hash}(x')) \wedge \mathsf{wf}(C) \wedge \mathsf{wf}(C') \wedge v < v' \implies x = x'.$$

Proof Fix v, C and x, and assume $\mathsf{committed}(C, v, \mathsf{hash}(x))$ and $\mathsf{wf}(C)$. We prove by induction on v' that

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

Assume this holds for all $v' < v^*$; we now prove it for $v' = v^*$. To this end, assume $v < v'$ and $\mathsf{prepared}(C', v', \mathsf{hash}(x'))$ for a well-formed C'.

Since $\mathsf{committed}(C, v, \mathsf{hash}(x))$ and $\mathsf{wf}(C)$, a quorum Q of processes sent $\mathtt{COMMITTED}(v, \mathsf{hash}(x))$. Since $\mathsf{prepared}(C', v', \mathsf{hash}(x'))$ and $\mathsf{wf}(C')$, a quorum Q' of processes sent $\mathtt{PREPARED}(v', \mathsf{hash}(x'))$. The quorums Q and Q' have to intersect in some correct process p_k, which has thus sent both $\mathtt{COMMITTED}(v, \mathsf{hash}(x))$ and $\mathtt{PREPARED}(v', \mathsf{hash}(x'))$. Since $v < v'$, process p_k must have sent $\mathtt{COMMITTED}(v, \mathsf{hash}(x))$ before $\mathtt{PREPARED}(v', \mathsf{hash}(x'))$. Before sending $\mathtt{COMMITTED}(v, \mathsf{hash}(x))$ the process set $\mathsf{locked\_view}$ to v (line 24) and had $\mathsf{prepared\_val} = x$.

Assume towards a contradiction that $x \neq x'$. Let v'' be the first view after v when p_k prepared some proposal $x'' \neq x$, so that $v'' \leq v'$. When this happened, by Proposition 8 process p_k must have had $\mathsf{prepared\_val} = x$ and $\mathsf{locked\_view} \geq v$. Then by the $\mathsf{SafeProposal}$ check (line 12), the leader of v'' provided a well-formed prepared certificate C'' such that $\mathsf{prepared}(C'', v''', \mathsf{hash}(x''))$ for v''' such that $v < v''' < v'' \leq v'$. But then by induction hypothesis we have $x'' = x$, and above we established $x'' \neq x$: a contradiction. Hence, we must have $x = x'$, as required. □

Corollary 14 *Single-shot HotStuff satisfies Agreement.*

Proof Analogous to the proof of Corollary 9, but using Lemma 16 instead of Lemma 15. □

Liveness and latency Assume that the protocol is used with a synchronizer satisfying Properties 1–5 in Fig. 2; 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 4 *Consider an execution of single-shot HotStuff with an eventual message delay bound δ, and let $v \geq \mathcal{V}$ be a view such that $F(v) \geq 7\delta$ and $\mathsf{leader}(v)$ is correct. Then all correct processes decide in view v by $E_{\mathrm{last}}(v) + 5\delta$.*

Proof Once a correct process enters v, it sends its $\mathtt{NEWLEADER}$ message, so that $\mathsf{leader}(v)$ is guaranteed to receive a quorum of such messages by $E_{\mathrm{last}}(v)+\delta$. When this happens, the leader will send its proposal in a $\mathtt{PROPOSE}$ message, which correct processes will receive by $E_{\mathrm{last}}(v) + 2\delta$. If they deem the proposal safe, it takes them at most 3δ to exchange the sequence of $\mathtt{PREPARED}$, $\mathtt{PRECOMMITTED}$ and $\mathtt{COMMITTED}$ messages leading to decisions. By (2), all correct processes will stay in v until at least $E_{\mathrm{last}}(v)+(F(v)-d) \geq E_{\mathrm{last}}(v) + 5\delta$, and thus will not send a message with a view $> v$ until this time. Then 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 $E_{\mathrm{last}}(v) + 5\delta$.

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 12). 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 $\mathtt{PROPOSE}$ message from the leader.

We now consider the case when some correct process is locked on a value when receiving the $\mathtt{PROPOSE}$ 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 $\mathtt{PRECOMMITTED}(v_0, \mathsf{hash}(x))$ from a quorum of processes (line 22), at least $f+1$ of which have to be correct. The latter processes must have assembled a prepared certificate for the value x at view v_0 (line 16). By Proposition 8, 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 $\mathtt{NEWLEADER}(v, \ldots)$ 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 $2f + 1$ $\mathtt{NEWLEADER}$ 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 4 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. □

Corollary 15 *Consider an execution with an eventual message delay bound δ, and assume that (1) holds for $u = 7\delta$. Then in this execution single-shot HotStuff satisfies Termination.*

Similarly to Sect. 5.1, 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 5 below uses Property C and Proposition 2, and the following two corollaries, Property B. Like in Sect. 5.2, the last corollary takes into account the optimized execution of view 1.

Corollary 16 *Let $v = \mathsf{GV}(\mathsf{GST} + \rho) + 1$ and assume that $S_{\mathrm{first}} < \mathsf{GST}$, $F(v) \geq 7\delta$ and $S_{f+1} \leq \mathsf{GST} + \rho$. Then in* *single-shot HotStuff all correct processes decide by* GST $+ \rho + \sum_{k=v-1}^{v+f-1}(F(k)+\delta)+7\delta$.

Corollary 17 *Assume that* $S_{\text{first}} \geq$ GST *and* $F(1) \geq 7\delta$. *Then in single-shot HotStuff all correct processes decide by* $S_{\text{last}} + \sum_{k=1}^{f}(F(k)+\delta)+6\delta$.

Corollary 18 *Assume that* $S_{\text{first}} \geq$ GST, $F(1) \geq 6\delta$, *and* leader(1) *is correct. Then in single-shot HotStuff all correct processes decide by* $S_{\text{last}} + 5\delta$.

Assume that F is such that (3) holds for $U = 7\Delta$. Then by Proposition 1 and Corollary 15 single-shot HotStuff with FastSync is live. Furthermore, under the conditions of Corollaries 16 and 17 its latency is bounded by GST $+ \rho + (7\Delta+\delta)(f+1)+7\delta$ if starting before GST, and by $S_{\text{last}} + (7\Delta+\delta)f+6\delta$ if starting after GST. As expected, the latency bounds for HotStuff are higher than those for PBFT (Sect. 5.1) due to an extra message exchange.

Communication complexity To reach a decision in a single view with a correct leader, the protocol requires exchanging messages with $O(n^2)$ signatures. Using threshold signatures, this communication complexity can be lowered to $O(n)$ [45]. To this end, instead of performing an all-to-all message exchange processes can send partial signatures to the leader, which aggregates them into a single threshold signature and distributes it to processes. Our results can be easily adjusted to this variant of the protocol.

5.3 Two-phase HotStuff

We next consider a *two-phase* variant of HotStuff [45], which processes the leader's proposals in two phases instead of three (Algorithm 4). To keep the same communication complexity as the three-phase HotStuff, the two-phase version uses timeouts not just for view synchronization, but also in the core consensus protocol to delimit different stages of a single view. This is similar to Tendermint [14] and Casper [15], which use timeouts for the same purposes. By handling two-phase HotStuff we demonstrate that our synchronizer specification is strong enough to deal with interactions between the timeouts in different parts of the overall protocol. We also show that the protocol requires only bounded space when used with our FastSync synchronizer.

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 14 in Algorithm 4). Upon assembling a prepared certificate for a value x (line 19), a process stores the value in prepared_val and the certificate in cert. In contrast to HotStuff, the process then immediately becomes locked on the value, without exchanging PRECOMMITTED messages; this is recorded by assigning locked_view to the current view (the prepared_view variable is thus unnecessary). The process then sends a

```
1  upon new_view($v$)
2      curr_view ← $v$;
3      voted ← FALSE;
4      stop_timer(timer_newleader);
5      if $p_i$ = leader(curr_view) then
6          start_timer(timer_newleader,
               $F_p$(curr_view));
7      send ⟨NEWLEADER(curr_view, locked_view,
           prepared_val, cert)⟩$_i$ to leader(curr_view);

8  when timer_newleader expired and received $M$ =
   $\{\langle$NEWLEADER$(v, view_j, val_j, cert_j)\rangle_j \mid p_j \in P\}$
9      pre: curr_view = $v$ ∧ $p_i$ = leader($v$) ∧
           ($\forall m \in M$. ValidNewLeader($m$));
10     if $\exists j.\ view_j = \max\{view_k \mid p_k \in P\} \neq 0$ then
11         send ⟨PROPOSE($v$, $val_j$, $cert_j$)⟩$_i$ to all;
12     else
13         send ⟨PROPOSE($v$, myval(), ⊥)⟩$_i$ to all;

14 when received ⟨PROPOSE($v$, $x$, _)⟩$_j$ = $m$
15     pre: curr_view = $v$ ∧ voted = FALSE ∧
           SafeProposal($m$);
16     curr_val ← $x$;
17     voted ← TRUE;
18     send ⟨PREPARED($v$, hash(curr_val))⟩$_i$ to all;

19 when received $\{\langle$PREPARED$(v, h)\rangle_j \mid p_j \in Q\} = C$
   for a quorum $Q$
20     pre: curr_view = $v$ ∧ voted = TRUE ∧
           hash(curr_val) = $h$;
21     prepared_val ← curr_val;
22     locked_view ← curr_view;
23     cert ← $C$;
24     send ⟨COMMITTED($v$, $h$)⟩$_i$ to all;

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

Algorithm 4: Two-phase HotStuff at a process p_i. All variables storing views are initially set to 0 and others to ⊥. The predicates ValidNewLeader and SafeProposal are defined by (16) and (20).

COMMITTED message with the hash of the locked value, and assembling a quorum of such messages causes the process to decide on the value (line 25). 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 7). The leader computes its proposal from NEWLEADER messages in the same way as in three-phase HotStuff (line 10).

The two-phase version of HotStuff satisfies Validity and Agreement for the same reasons as the three-phase one: as we noted in Sect. 5.2, 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 only after the timer expires (line 8).

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 5 *Consider an execution of two-phase HotStuff with an eventual message delay bound δ, and let $v \geq \mathcal{V}$ be a view such that $F_p(v) \geq 3\delta$, $F(v) - F_p(v) \geq 5\delta$ (so that $F(v) \geq 8\delta$) and* leader(v) *is correct. Then all correct processes decide in view 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 of the synchronizer, 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 $F_p(v) \geq 3\delta$, 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 $E_{\text{last}}(v) + (F(v) - d) \geq E_{\text{first}}(v) + F_p(v) + 3\delta$. Then 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 $E_{\text{last}}(v) + F_p(v) + 3\delta$.

It remains to show that the proposal leader(v) makes in view v (line 8) will satisfy SafeProposal at all correct processes (line 14). It is easy to show that this proposal satisfies 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 4, $x' = x$ 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. □

Corollary 19 *Consider an execution with an eventual message delay bound δ, and assume that F is such that (1) holds respectively for F and $u = 8\delta$, and for $F - F_p$ and $u = 5\delta$. Then in this execution two-phase HotStuff satisfies Termination.*

Thus, one way to ensure the liveness of two-phase HotStuff is to pick functions F and F_p such that both of these, *as well as the difference between them*, grow without bound. This can be satisfied, e.g., by letting $F(v) = 2v$ and $F_p(v) = v$. By Proposition 1, another way to ensure liveness is to pick F such that (3) holds for $U = 8\Delta$, and F_p is such that the same property holds for $U = 3\Delta$; this option prevents the timeouts from growing unboundedly.

Latency The following corollaries of Theorem 4 quantify the latency of two-phase HotStuff with the with the FastSync synchronizer in different scenarios. The first corollary uses Property C and Proposition 2, and the following two corollaries, Property B. Like in Sect. 5.1, the last corollary takes into account the optimized execution of view 1.

Corollary 20 *Let $v = \mathsf{GV}(\mathsf{GST} + \rho) + 1$ and assume that $S_{\text{first}} < \mathsf{GST}$, $S_{f+1} \leq \mathsf{GST} + \rho$, $F_p(v) \geq 3\delta$ and $F(v) - F_p(v) \geq 5\delta$. Then in two-phase HotStuff all correct processes decide by $\mathsf{GST} + \rho + \sum_{k=v-1}^{v+f-1}(F(k) + \delta) + F_p(v + f) + 5\delta$.*

Corollary 21 *Assume that $S_{\text{first}} \geq \mathsf{GST}$, $F_p(1) \geq 3\delta$ and $F(1) - F_p(1) \geq 5\delta$. Then in two-phase HotStuff all correct processes decide by $S_{\text{last}} + \sum_{k=1}^{f}(F(k)+\delta)+F_p(f+1)+4\delta$.*

Corollary 22 *Assume that $S_{\text{first}} \geq \mathsf{GST}$, $F(1) \geq 5\delta$ and $\mathsf{leader}(1)$ is correct. Then in two-phase HotStuff all correct processes decide by $S_{\text{last}} + 4\delta$.*

Assume that F is such that (3) holds for $U = 8\Delta$, and F_p is such that the same property holds for $U = 3\Delta$; then as we noted above, two-phase HotStuff with FastSync is live. Furthermore, under the conditions of Corollaries 20 and 21 its latency is bounded by $\mathsf{GST}+\rho+(8\Delta+\delta)(f+1)+3\Delta+5\delta$ if starting before GST, and by $S_{\text{last}} + (8\Delta + \delta)f + 3\Delta + 4\delta$ if starting after GST.

The latency bounds we established allow us to compare the two-phase version of HotStuff with the its three-phase version (Sect. 5.2) and PBFT (Sect. 5.1). In the ideal case when the network is synchronous, the timeouts are set to the minimal values ensuring liveness, and the leader of view 1 is correct, two-phase HotStuff has the same latency as PBFT and a lower latency than three-phase HotStuff: 4δ in Corollaries 13 and 22 vs 5δ in Corollary 18. When the initial leader is faulty, all protocols incur the overhead of switching through several views until they encounter a correct leader (Corollaries 12, 17 and 21). In this case, the latency of deciding in the first view with a correct leader is at most 6δ for three-phase HotStuff, 5δ for PBFT and $F_p(f+1)+4\delta$ for two-phase one. In the worst case the latency of two-phase HotStuff is $F_p(f+1)+4\delta \leq 3\Delta+4\delta$. This is much higher than the latency of the other protocols, since in practice Δ gives only a conservative estimate of the message delay. But 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 and PBFT. The latency bounds for the case of starting before GST relate similarly (Corollaries 11, 16 and 20). 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 [45].

5.4 Summary and additional case studies

We have shown that our synchronizer specification is strong enough to guarantee liveness under partial synchrony for three Byzantine consensus protocols using different approaches:

- In PBFT the leader proposals go through two phases of message exchanges. Both safety and liveness are ensured by the leader sending supporting information together with its proposal that allows processes to verify the proposal's correctness.
- HotStuff lowers the communication complexity by reducing the amount of supporting information the leader has to send with its proposal. Achieving liveness then requires an additional message exchange to ensure that the leader has an up-to-date information when making its proposal.
- Two-phase HotStuff eliminates the extra message exchange of HotStuff and instead ensures liveness using a timeout within a view: the leader waits until it receives the information from all correct processes before making its proposal.

To further demonstrate the wide applicability of our synchronizer specification, we have also used it to prove the correctness and analyze the latency of single-shot versions of SBFT [31] and Tendermint [14]. We defer the details to [12, §B]. 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 processes leader proposals in two phases and 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 Sect. 5.3. 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 [6,14], which lacked a detailed correctness argument for the view synchronization mechanism used in the protocol.

6 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 [14,19,31]. Subtle view synchronization mechanisms have often come without a proof of liveness (e.g., PBFT [18]) or had liveness bugs (e.g., Tendermint [5] and Casper [35]). Furthermore, liveness proofs have not usually given concrete bounds on the latency of reaching a decision (exceptions are [4,39]).

Several papers suggested separating the functionality of view synchronization into a distinct component, including the seminal DLS paper on consensus under partial synchrony [27] and its more modern implementation [25]. 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. The model guarantees that, eventually, a process in a round r receives all messages sent by correct processes in r. This property is needlessly strong for Byzantine consensus, since protocols such as PBFT (Sect. 5.1) and HotStuff (Sect. 5.2) can make progress in a given view if they receive messages from *any* quorum; executing such protocols in the DLS model would thus undermine their optimistic responsiveness [45]. 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. [1] build upon ideas from fault-tolerant clock synchronization [24,43] 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 [45] 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 [41,42]. 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 Fig. 2), which are necessary for protocols such as two-phase HotStuff (Sect. 5.3) and Tendermint (Sect. 5.4).

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 $\mathsf{GST}+\delta$ [41,42]. These implementations focus on optimizing communication complexity, making it linear in best-case scenarios [41] or in expectation [42]. 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 Sect. 3.1). Finally, we give exact latency bounds for FastSync under both favorable and unfavorable conditions whereas [41,42] 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.

DiemBFT [44] extends HotStuff with a view synchronization mechanism, integrated with the core protocol; the protocol assumes reliable channels. DiemBFT is optimized to solve repeated consensus, whereas in this paper we focus on single-shot one.

The original idea of using synchronizers to simulate a round-based synchronous system on top of an asynchronous one is due to Awerbuch [8]. 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 [10,22,29,36]. 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 [3,26]. While we derive similar bounds for Byzantine failures, it remains open if these are optimal or can be further improved. Failure detectors [20,21], which abstract away the timeliness guarantees of the environment, have been extensively used for developing and analyzing consensus algorithms [21,40] 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., [38]), or focuses on other means (such as accountability [32]) to mitigate faulty behavior.

The generalized partial synchrony model stipulating the existence of a fixed but unknown post-GST message delay bound δ is due to [21]. The time complexity measure combining δ with an a priori known conservative message delay bound $\Delta \geq \delta$ was first introduced in [33,34]. This work, however, assumed a stronger variant of partial synchrony where δ is unknown but holds throughout the entire execution, rather than eventually [27]. We adopt a similar metric for our latency analysis, but only require that the two bounds δ and Δ hold after GST. To the best of our knowledge, our analysis of Byzantine consensus latency is the first one in this model.

7 Conclusion

We have proposed a modular approach for ensuring liveness in Byzantine consensus under a general variant of the partial synchrony model, which permits unlimited message loss and out-of-sync clocks before GST, and does not stipulate the knowledge of communication delay bounds after GST. To this end, we have proposed a specification of a view synchronizer that is strong enough to ensure liveness in a number of well-known Byzantine consensus algorithms. We have also shown that this specification is implementable under partial synchrony in bounded space, despite message loss before GST. We believe that our synchronizer abstraction provides a much needed tool for facilitating the design of partially synchronous Byzantine consensus protocols, and for enabling a systematic analysis of their performance guarantees. In our current work we are generalizing the results presented here from (single-shot) Byzantine consensus to Byzantine state-machine replication [13].

Acknowledgements Alexey Gotsman was supported by a Starting Grant RACCOON from the European Research Council.

Appendix A: General latency bounds

We now augment the set of properties in Fig. 2 with two additional latency bounds given by Properties D and E in Fig. 3. These two properties are analogous to Properties B and C, but handle the cases when $F(1) < 2\delta$ and $F(\mathsf{GV}(\mathsf{GST} + \rho) + 1) < 2\delta$, respectively. We then prove Theorem 6, which generalizes Theorem 1 to establish that FastSync also satisfies Properties D and E in Fig. 3 in addition to those given in Fig. 2.

The resulting set of properties can be used to derive latency bounds for specific instantiations of the timeout function F. We demonstrate how to do this for an exponentially growing timeout function, which is a common choice in practice (e.g., [19]). Specifically, we prove that if $\forall v > 0.\ F(v) = 2^v$, then all correct processes are guaranteed to enter a synchronized view within $O(\delta \lg \delta)$ after S_{last} if the protocol is started after GST (Theorem 7); and within $O(\max\{\delta \lg \delta, \Delta\})$ after $\mathsf{GST}+\rho$, otherwise (Corollary 23). The latter guarantees that the latency of view synchronization is bounded after GST.

Theorem 6 *Consider an execution with an eventual message delay bound δ, and assume that (1) holds for $u = 2\delta$. Then there exists a view $\mathcal{V}$ such that in this execution* FastSync *satisfies all the properties in Figs. 2 and 3 for $d = 2\delta$.*

To prove the Theorem 6, we first prove the following proposition, which is an easy consequence of the definition of GV.

Proposition 9 *For all views v, if a correct process enters v, then $\mathsf{GV}(E_{\text{first}}(v)) = v$.*

Proof By definition of GV, $\mathsf{GV}(E_{\text{first}}(v)) \geq v$. Assume by contradiction that $\mathsf{GV}(E_{\text{first}}(v)) > v$. Thus, there exists a correct process that enters a view $v' > v$ at time $t' < E_{\text{first}}(v)$. By Lemma 1, there exists a time $t < t'$ such that some correct process enters v at t. Thus, $E_{\text{first}}(v) \leq t < t' < E_{\text{first}}(v)$, which is a contradiction. □

The next lemma generalizes Corollary 7 to bound the latency of entering an arbitrary view $\geq \mathsf{GV}(t) + 1$ for all $t \geq \overline{\mathsf{GST}}$ and $\mathsf{GV}(t) > 0$.

Lemma 17 *Let $t \geq \overline{\mathsf{GST}}$ and suppose that $\mathsf{GV}(t) > 0$. If for all $k \geq 1$, some correct process enters every view $\mathsf{GV}(t)+k$, then $E_{\text{last}}(\mathsf{GV}(t) + k) \leq t + \sum_{i=0}^{k-1} F(\mathsf{GV}(t) + i) + 3k\delta$.*

Proof By induction on $k \geq 1$. Since some correct process enters $\mathsf{GV}(t) + 1$ and $\mathsf{GV}(t) > 0$, Corollary 7 implies that $E_{\text{last}}(\mathsf{GV}(t) + 1) \leq t + F(\mathsf{GV}(t)) + 3\delta$. Thus, the required holds for the base case of $k = 1$.

For the inductive step, assume that the required holds for $k = l$ where $l \geq 1$, and consider $k = l + 1$. Suppose that

$$t \geq \overline{\mathsf{GST}} \wedge \mathsf{GV}(t) > 0.$$

Then, by the induction hypothesis,

$$E_{\text{last}}(\mathsf{GV}(t) + l) \leq t + \sum_{i=0}^{l-1} F(\mathsf{GV}(t) + i) + 3l\delta. \tag{21}$$

Since some correct process enters $\mathsf{GV}(t)+l$, $E_{\text{first}}(\mathsf{GV}(t)+l)$ is defined. Thus, by Lemma 9, we have

$$\mathsf{GV}(E_{\text{first}}(\mathsf{GV}(t) + l)) = \mathsf{GV}(t) + l > \mathsf{GV}(t) > 0.$$

Since GV is non-decreasing, the above implies that

$$E_{\text{first}}(\mathsf{GV}(t) + l) > t \geq \overline{\mathsf{GST}}.$$

Thus, by Corollary 7,

$$\begin{aligned} &E_{\text{last}}(\mathsf{GV}(t) + l + 1) \\ &\quad \leq E_{\text{first}}(\mathsf{GV}(t) + l) + F(\mathsf{GV}(t) + l) + 3\delta \\ &\quad \leq E_{\text{last}}(\mathsf{GV}(t) + l) + F(\mathsf{GV}(t) + l) + 3\delta, \end{aligned}$$

which by (21), implies

$$\begin{aligned} &E_{\text{last}}(\mathsf{GV}(t) + l + 1) \\ &\quad \leq E_{\text{last}}(\mathsf{GV}(t) + l) + F(\mathsf{GV}(t) + l) + 3\delta \\ &\quad \leq t + \sum_{i=0}^{l-1} F(\mathsf{GV}(t) + i) + 3l\delta + F(\mathsf{GV}(t) + l) + 3\delta \\ &\quad = t + \sum_{i=0}^{(l+1)-1} F(\mathsf{GV}(t) + i) + 3(l + 1)\delta, \end{aligned}$$

as required. □

Proof of Theorem 6. Consider an execution of FastSync and let δ be the eventual message delay bound in this execution.

D. $S_{\text{first}} \geq \mathsf{GST} \wedge F(1) < 2\delta \implies$
$\mathcal{V} = \min\{v \mid F(v) \geq 2\delta\} \wedge$
$E_{\text{last}}(\mathcal{V}) \leq S_{\text{last}} + \sum_{i=1}^{\mathcal{V}-1} F(i) + (3\mathcal{V} - 2)\delta$

E. $S_{\text{first}} < \mathsf{GST} \wedge S_{f+1} \leq \mathsf{GST} + \rho \wedge$
$F(\mathsf{GV}(\mathsf{GST} + \rho) + 1) < 2\delta \implies$
$\mathcal{V} = \min\{v \mid F(v) \geq 2\delta\} \wedge$
$E_{\text{last}}(\mathcal{V}) \leq \mathsf{GST} + \rho + \sum_{i=0}^{\mathcal{V}-1} F(i) + 3\mathcal{V}\delta$

Fig. 3 Additional FASTSYNC latency bounds

We first show how to select a view $\mathcal{V}$ such that (9) holds. We consider the following cases. First, if

$$S_{\text{first}} \geq \mathsf{GST} \wedge F(1) \geq 2\delta, \tag{22}$$

then we let $\mathcal{V} = 1$. Since $S_{\text{first}} \geq \mathsf{GST}$, the definition of $\overline{\mathsf{GST}}$ implies $\overline{\mathsf{GST}} = S_{\text{first}}$, and therefore, (9) holds. Second, if

$$S_{\text{first}} < \mathsf{GST} \;\wedge\; S_{f+1} \;\leq\; \mathsf{GST} + \rho \wedge \\ F(\mathsf{GV}(\mathsf{GST} + \rho) + 1) \geq 2\delta, \tag{23}$$

then we let $\mathcal{V} = \mathsf{GV}(\mathsf{GST} + \rho) + 1$. Since $S_{\text{first}} < \mathsf{GST}$, the definition of $\overline{\mathsf{GST}}$ implies $\overline{\mathsf{GST}} = \mathsf{GST} + \rho$, and therefore, (9) holds. Third, if

$$S_{\text{first}} \geq \mathsf{GST} \wedge F(1) < 2\delta, \tag{24}$$

then we let $\mathcal{V} = \min\{v \mid F(v) \geq 2\delta\}$. Since $S_{\text{first}} \geq \mathsf{GST}$, the definition of $\overline{\mathsf{GST}}$ implies $\overline{\mathsf{GST}} = S_{\text{first}}$. By the monotonicity of F, $F(1) < 2\delta$ implies that $\mathcal{V} > 1 = \mathsf{GV}(S_{\text{first}}) + 1$. Thus, (9) holds. Fourth, if

$$S_{\text{first}} < \mathsf{GST} \;\wedge\; S_{f+1} \;\leq\; \mathsf{GST} + \rho \;\wedge \\ F(\mathsf{GV}(\mathsf{GST} + \rho) + 1) < 2\delta, \tag{25}$$

then we let $\mathcal{V} = \min\{v \mid F(v) \geq 2\delta\}$. Since $S_{\text{first}} < \mathsf{GST}$, the definition of $\overline{\mathsf{GST}}$ implies $\overline{\mathsf{GST}} = \mathsf{GST} + \rho$. By the monotonicity of F, $F(\mathsf{GV}(\mathsf{GST} + \rho) + 1) < 2\delta$ implies that $\mathcal{V} > \mathsf{GV}(\mathsf{GST} + \rho) + 1$, and therefore, (9) holds. In all other cases, (1) for $u = 2\delta$ implies that $F(v') \geq 2\delta$ for some view v', and therefore, by the monotonicity of F, $\mathcal{V} = \max\{v', \mathsf{GV}(\overline{\mathsf{GST}}) + 1\}$ satisfies (9). Thus, by Lemma 10, Properties 1–5 in Fig. 2 hold for $\mathcal{V}$ chosen as above and $d = 2\delta$.

To prove Property A, fix $v \geq \mathcal{V}$. By Property 3, all correct processes enter v. By (9), $v \geq \mathcal{V} \geq \mathsf{GV}(\overline{\mathsf{GST}}) + 1$. Given that GV is non-decreasing, this implies that no correct process can enter v until after $\overline{\mathsf{GST}}$. Thus, $E_{\text{last}}(v) \geq E_{\text{first}}(v) > \overline{\mathsf{GST}}$, and by Corollary 4 we get $E_{\text{last}}(v + 1) \leq E_{\text{last}}(v) + F(v) + \delta$, validating Property A. Next, by our choice of $\mathcal{V}$, (22) implies $\mathcal{V} = 1$, and (23) implies $\mathcal{V} = \mathsf{GV}(\mathsf{GST} + \rho) + 1$. Thus, Property B follows from Corollary 6, and Property C from Corollary 7.

We now prove Property D in Fig. 3. By our choice of $\mathcal{V}$, (24) implies that $\mathcal{V} = \min\{v \mid F(v) \geq 2\delta\}$. By Lemma 9, $\mathsf{GV}(E_{\text{first}}(1)) = 1$. Since $F(1) < 2\delta$, the monotonicity of F implies $\mathcal{V} > 1$, and therefore,

$$\exists k \geq 1.\, \mathcal{V} = 1 + k. \tag{26}$$

Instantiating Lemma 17 with $t = E_{\text{first}}(1) > S_{\text{first}} = \overline{\mathsf{GST}}$ and $\mathsf{GV}(E_{\text{first}}(1)) = 1 > 0$, we get

$$E_{\text{last}}(1 + k) \leq E_{\text{first}}(1) + \sum_{i=0}^{k-1} F(1 + i) + 3k\delta,$$

which by (26) implies

$$E_{\text{last}}(\mathcal{V}) \leq E_{\text{first}}(1) + \sum_{i=0}^{\mathcal{V}-2} F(1 + i) + 3(\mathcal{V} - 1)\delta \\ \leq E_{\text{last}}(1) + \sum_{i=0}^{\mathcal{V}-2} F(1 + i) + 3(\mathcal{V} - 1)\delta.$$

Hence, by Corollary 6, we have

$$E_{\text{last}}(\mathcal{V}) \leq S_{\text{last}} + \delta + \sum_{i=0}^{\mathcal{V}-2} F(i + 1) + 3(\mathcal{V} - 1)\delta \\ = S_{\text{last}} + \sum_{i=1}^{\mathcal{V}-1} F(i) + (3\mathcal{V} - 2)\delta, \tag{27}$$

as required.

Lastly, we prove Property E in Fig. 3. By our choice of $\mathcal{V}$, (25) implies that $\mathcal{V} = \min\{v \mid F(v) \geq 2\delta\}$. By Lemma 9, $\mathsf{GV}(E_{\text{first}}(\mathsf{GV}(\mathsf{GST} + \rho) + 1)) = \mathsf{GV}(\mathsf{GST} + \rho) + 1$. Since $F(\mathsf{GV}(\mathsf{GST} + \rho) + 1) < 2\delta$, the monotonicity of F implies $\mathcal{V} > \mathsf{GV}(\mathsf{GST} + \rho) + 1$, and therefore,

$$\exists k \geq 1.\, \mathcal{V} = \mathsf{GV}(\mathsf{GST} + \rho) + 1 + k. \tag{28}$$

Instantiating Lemma 17 with $t = E_{\text{first}}(\mathsf{GV}(\mathsf{GST} + \rho) + 1) > \mathsf{GST} + \rho = \overline{\mathsf{GST}}$ and $\mathsf{GV}(E_{\text{first}}(\mathsf{GV}(\mathsf{GST} + \rho) + 1)) = \mathsf{GV}(\mathsf{GST} + \rho) + 1 > 0$, we get

$$E_{\text{last}}(\mathsf{GV}(\mathsf{GST} + \rho) + 1 + k) \\ \leq E_{\text{first}}(\mathsf{GV}(\mathsf{GST} + \rho) + 1) \\ + \sum_{i=0}^{k-1} F(\mathsf{GV}(\mathsf{GST} + \rho) + 1 + i) + 3k\delta,$$

which by (28) implies

$$\begin{aligned}
&E_{\text{last}}(\mathcal{V}) \le E_{\text{first}}(\mathsf{GV}(\mathsf{GST}+\rho)+1)\\
&\quad+\sum_{i=0}^{\mathcal{V}-\mathsf{GV}(\mathsf{GST}+\rho)-2} F(\mathsf{GV}(\mathsf{GST}+\rho)+1+i)\\
&\quad+3(\mathcal{V}-\mathsf{GV}(\mathsf{GST}+\rho)-1)\delta \le\\
&E_{\text{last}}(\mathsf{GV}(\mathsf{GST}+\rho)+1)\\
&\quad+\sum_{i=0}^{\mathcal{V}-\mathsf{GV}(\mathsf{GST}+\rho)-2} F(\mathsf{GV}(\mathsf{GST}+\rho)+1+i)\\
&\quad+3(\mathcal{V}-\mathsf{GV}(\mathsf{GST}+\rho)-1)\delta.
\end{aligned}$$

Hence, by Corollary 7, we have

$$\begin{aligned}
&E_{\text{last}}(\mathcal{V}) \le E_{\text{last}}(\mathsf{GV}(\mathsf{GST}+\rho)+1)\\
&\quad+\sum_{i=0}^{\mathcal{V}-\mathsf{GV}(\mathsf{GST}+\rho)-2} F(\mathsf{GV}(\mathsf{GST}+\rho)+1+i)\\
&\quad+3(\mathcal{V}-\mathsf{GV}(\mathsf{GST}+\rho)-1)\delta\\
&\quad\le \mathsf{GST}+\rho+F(\mathsf{GV}(\mathsf{GST}+\rho))+3\delta\\
&\quad+\sum_{i=0}^{\mathcal{V}-\mathsf{GV}(\mathsf{GST}+\rho)-2} F(\mathsf{GV}(\mathsf{GST}+\rho)+1+i)\\
&\quad+3(\mathcal{V}-\mathsf{GV}(\mathsf{GST}+\rho)-1)\delta\\
&\quad= \mathsf{GST}+\rho+\sum_{i=0}^{\mathcal{V}-\mathsf{GV}(\mathsf{GST}+\rho)-1} F(\mathsf{GV}(\mathsf{GST}+\rho)+i)\\
&\quad+3(\mathcal{V}-\mathsf{GV}(\mathsf{GST}+\rho))\delta\\
&\quad\le \mathsf{GST}+\rho+\sum_{i=0}^{\mathcal{V}-1} F(i)+3\mathcal{V}\delta,
\end{aligned}$$

as required. □

We now use Theorem 6 to derive closed-form expressions for view $\mathcal{V}$ and the latency of reaching it after GST assuming an exponentially growing timeout function $F(v) = 2^v$ for all $v > 0$. Below we show that if the protocol starts after GST ($\mathcal{S}_{\text{first}} \ge \mathsf{GST}$), then all correct processes are guaranteed to synchronize in the view $\mathcal{V} = \max\{\lceil \lg 2\delta \rceil, 1\}$, which they enter within $O(\delta \lg \delta)$ of the last correct process calling `start`.

Theorem 7 *Consider an execution of with an eventual message delay bound δ, and assume that (1) holds for $u = 2\delta$, $\mathcal{S}_{\text{first}} \ge \mathsf{GST}$, and $\forall v > 0.\ F(v) = 2^v$. Then in this execution* FastSync *satisfies all the properties in Figs. 2 and 3 for $\mathcal{V} = \max\{\lceil \lg 2\delta \rceil, 1\}$ and $d = 2\delta$. Furthermore, it holds:*

$$E_{\text{last}}(\mathcal{V}) \le \mathcal{S}_{\text{last}} + 3\delta \lg \delta + 8\delta = \mathcal{S}_{\text{last}} + O(\delta \lg \delta).$$

Proof We consider two cases. Suppose first that $F(1) \ge 2\delta$. Then, by Theorem 6, all the properties in Figs. 2 and 3 hold for $\mathcal{V} = 1$ and $d = 2\delta$, and by Property B,

$$E_{\text{last}}(\mathcal{V}) \le \mathcal{S}_{\text{last}} + \delta. \tag{29}$$

Since $F(1) \ge 2\delta$, by our choice of the timeout function, $2^1 \ge 2\delta$. Hence, $\lg 2\delta \le 1$, and therefore, $\lceil \lg 2\delta \rceil \in \{0, 1\}$. Thus, we get $\mathcal{V} = 1 = \max\{\lceil \lg 2\delta \rceil, 1\}$, as needed.

Suppose next that $F(1) < 2\delta$. Then, by Theorem 6, all the properties in Figs. 2 and 3 hold for $\mathcal{V} = \min\{v \mid F(v) \ge 2\delta\}$ and $d = 2\delta$, and by Property D,

$$E_{\text{last}}(\mathcal{V}) \le \mathcal{S}_{\text{last}} + \sum_{i=1}^{\mathcal{V}-1} F(i) + (3\mathcal{V}-2)\delta. \tag{30}$$

Since $F(1) < 2\delta$, $\mathcal{V} > 1$, and therefore,

$$\begin{aligned}
\mathcal{V} &= \min\{v \mid 2^v \ge 2\delta \wedge v > 1\}\\
&= \min\{v \mid v \ge \lg 2\delta \wedge v > 1\},
\end{aligned}$$

which implies

$$\mathcal{V} = \max\{\lceil \lg 2\delta \rceil, 1\}, \tag{31}$$

as needed.

Finally, plugging (31) into (30) and using the fact that $\lceil \lg 2\delta \rceil \le \lg 2\delta + 1$, we get

$$\begin{aligned}
E_{\text{last}}(\mathcal{V}) &\le \mathcal{S}_{\text{last}} + \sum_{i=1}^{\lg 2\delta} 2^i + (3 \lg 2\delta + 1)\delta\\
&\le \mathcal{S}_{\text{last}} + 3\delta \lg \delta + 8\delta = \mathcal{S}_{\text{last}} + O(\delta \lg \delta).
\end{aligned} \tag{32}$$

Thus, from (29) and (32), we get

$$E_{\text{last}}(\mathcal{V}) \le 3\delta \lg \delta + 8\delta = \mathcal{S}_{\text{last}} + O(\delta \lg \delta),$$

as required. □

We now show that if some correct process calls `start` before GST ($\mathcal{S}_{\text{first}} < \mathsf{GST}$), then all correct processes are guaranteed to synchronize in the view $\mathcal{V} = \max\{\lceil \lg 2\delta \rceil, \mathsf{GV}(\mathsf{GST}) + \rho + 1\}$, which they enter within $O(\max\{\delta \lg \delta, F(\mathcal{V}-1)\})$ after $\mathsf{GST} + \rho$.

Theorem 8 *Consider an execution of with an eventual message delay bound δ, and assume that (1) holds for $u = 2\delta$, $\mathcal{S}_{\text{first}} < \mathsf{GST}$, and $\forall v > 0.\ F(v) = 2^v$. Then in this execution* FastSync *satisfies all the properties in Figs. 2 and 3 for $\mathcal{V} = \max\{\lceil \lg 2\delta \rceil, \mathsf{GV}(\mathsf{GST}+\rho)+1\}$ and $d = 2\delta$. Furthermore, it holds:*

$$E_{\text{last}}(\mathcal{V}) \le \mathsf{GST} + \rho + \max\{3\delta \lg \delta, F(\mathcal{V}-1)\} + 10\delta$$

$$= \mathsf{GST} + \rho + O(\max\{\delta \lg \delta, F(\mathcal{V} - 1)\}).$$

Proof We consider two cases. Suppose first that $F(\mathsf{GV}(\mathsf{GST}+\rho)+1) \geq 2\delta$. Then, by Theorem 6, all the properties in Figs. 2 and 3 hold for $\mathcal{V} = \mathsf{GV}(\mathsf{GST} + \rho) + 1$ and $d = 2\delta$, and by Property C,

$$E_{\text{last}}(\mathcal{V}) \leq \mathsf{GST} + \rho + F(\mathcal{V} - 1) + 3\delta. \tag{33}$$

Since $F(\mathsf{GV}(\mathsf{GST}+\rho)+1) \geq 2\delta$, by our choice of the timeout function, $2^{\mathsf{GV}(\mathsf{GST}+\rho)+1} \geq 2\delta$. Hence, $\lg 2\delta \leq \mathsf{GV}(\mathsf{GST}+\rho)+1$, and therefore, $\lceil \lg 2\delta \rceil \leq \mathsf{GV}(\mathsf{GST} + \rho) + 1$. Thus, we get $\mathcal{V} = \mathsf{GV}(\mathsf{GST} + \rho) + 1 = \max\{\lceil \lg 2\delta \rceil, \mathsf{GV}(\mathsf{GST} + \rho) + 1\}$, as needed.

Suppose next that $F(\mathsf{GV}(\mathsf{GST} + \rho) + 1) < 2\delta$. Then, by Theorem 6, all the properties in Figs. 2 and 3 hold for $\mathcal{V} = \min\{v \mid F(v) \geq 2\delta\}$ and $d = 2\delta$, and by Property E,

$$E_{\text{last}}(\mathcal{V}) \leq \mathsf{GST} + \rho + \sum_{i=0}^{\mathcal{V}-1} F(i) + 3\mathcal{V}\delta. \tag{34}$$

Since $F(\mathsf{GV}(\mathsf{GST} + \rho) + 1) < 2\delta$, $\mathcal{V} > \mathsf{GV}(\mathsf{GST} + \rho) + 1$, and therefore,

$$\begin{aligned}\mathcal{V} &= \min\{v \mid 2^v \geq 2\delta \wedge v > \mathsf{GV}(\mathsf{GST} + \rho) + 1\}\\ &= \min\{v \mid v \geq \lg 2\delta \wedge v > \mathsf{GV}(\mathsf{GST} + \rho) + 1\},\end{aligned}$$

and therefore,

$$\mathcal{V} = \max\{\lceil \lg 2\delta \rceil, \mathsf{GV}(\mathsf{GST} + \rho) + 1\}, \tag{35}$$

as needed.

Finally, plugging (35) into (34) and using the fact that $\lceil \lg 2\delta \rceil \leq \lg 2\delta + 1$, we get

$$\begin{aligned}E_{\text{last}}(\mathcal{V}) &\leq \mathsf{GST} + \rho + \sum_{i=1}^{\lg 2\delta} 2^i + 3(\lg 2\delta + 1)\delta\\ &\leq \mathsf{GST} + \rho + 3\delta \lg \delta + 10\delta\\ &= \mathsf{GST} + \rho + O(\delta \lg \delta).\end{aligned} \tag{36}$$

Thus, from (29) and (36), and since $F(\mathcal{V}-1)$ can be arbitrarily large, we get

$$\begin{aligned}E_{\text{last}}(\mathcal{V}) &\leq \mathsf{GST} + \rho + \max\{3\delta \lg \delta, F(\mathcal{V} - 1)\} + 10\delta\\ &= \mathsf{GST} + \rho + O(\max\{\delta \lg \delta, F(\mathcal{V} - 1)\}),\end{aligned}$$

as required. □

By Proposition 1, we can apply Theorem 8 if (3) holds for $U = 2\Delta$. Then $F(\mathcal{V}-1) \leq 2\Delta$, which implies the following:

Corollary 23 *Suppose that (3) holds for $U = 2\Delta$, and $\forall v > 0.\, F(v) = 2^v$. Then, every execution of* FAST-SYNC *with the eventual message delay δ such that $S_{\text{first}} <$ GST satisfies all the properties in Figs. 2 and 3 for $\mathcal{V} = \max\{\lceil \lg 2\delta \rceil, \mathsf{GV}(\mathsf{GST} + \rho) + 1\}$ and $d = 2\delta$, and it holds:*

$$\begin{aligned}E_{\text{last}}(\mathcal{V}) &\leq \mathsf{GST} + \rho + \max\{3\delta \lg \delta, 2\Delta\} + 10\delta\\ &= \mathsf{GST} + \rho + O(\max\{\delta \lg \delta, \Delta\}).\end{aligned}$$

References

1. Abraham, I., Devadas, S., Dolev, D., Nayak, K., Ren, L.: 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)
2. Abraham, I., Gueta, G., Malkhi, D., Alvisi, L., Kotla, R., Martin, J.: Revisiting fast practical Byzantine fault tolerance. arXiv:1712.01367 (2017)
3. Alistarh, D., Gilbert, S., Guerraoui, R., Travers, C.: How to solve consensus in the smallest window of synchrony. In: Symposium on Distributed Computing (DISC) (2008)
4. Amir, Y., Coan, B.A., Kirsch, J., Lane, J.: Prime: Byzantine replication under attack. IEEE Trans. Dependable Secure Comput. **8**(4), 564–577 (2011)
5. Amoussou-Guenou, Y., Pozzo, A. D., Potop-Butucaru, M., Tucci-Piergiovanni, S.: Correctness of Tendermint-core blockchains. In: Conference on Principles of Distributed Systems (OPODIS) (2018)
6. Amoussou-Guenou, Y., Pozzo, A. D., Potop-Butucaru, M., Tucci-Piergiovanni, S.: Dissecting tendermint. In: Conference on Networked Systems (NETYS) (2019)
7. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., Caro, A.D., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolic, M., Cocco, S. W., Yellick, J.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: European Conference on Computer Systems (EuroSys) (2018)
8. Awerbuch, B.: Complexity of network synchronization. J. ACM **32**(4), 804–823 (1985)
9. Bazzi, R. A., Ding, Y.: Non-skipping timestamps for Byzantine data storage systems. In: Symposium on Distributed Computing (DISC) (2004)
10. Biely, M., Widder, J., Charron-Bost, B., Gaillard, A., Hutle, M., Schiper,A.: Tolerating corrupted communication. In: Symposium on Principles of Distributed Computing (PODC) (2007)
11. Bracha, G.: Asynchronous Byzantine agreement protocols. Inf. Comput. **75**(2), 130–143 (1987)
12. Bravo, M., Chockler, G., Gotsman, A.: Making Byzantine consensus live (extended version). CoRR arXiv:2008.04167 (2020)
13. Bravo, M., Chockler, G., Gotsman, A.: Liveness and latency of Byzantine state-machine replication. In: Symposium on Distributed Computing (DISC) (2022)
14. Buchman, E., Kwon, J., Milosevic,Z.: The latest gossip on BFT consensus. arXiv:1807.04938 (2018)
15. Buterin, V., Griffith, V.: Casper the friendly finality gadget. arXiv:1710.09437 (2017)
16. Cachin, C., Kursawe, K., Petzold, F., Shoup, V.: Secure and efficient asynchronous broadcast protocols. In: International Cryptology Conference (CRYPTO) (2001)

17. Cachin,C., Vukolic, M.: Blockchain consensus protocols in the wild (keynote talk). In: *Symposium on Distributed Computing (DISC)* (2017)
18. Castro, M.: Practical Byzantine fault tolerance. PhD thesis, Massachusetts Institute of Technology (2001)
19. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. In: Symposium on Operating Systems Design and Implementation (OSDI) (1999)
20. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM **43**(4), 685–722 (1996)
21. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
22. Charron-Bost, B., Schiper, A.: The heard-of model: computing in distributed systems with benign faults. Distrib. Comput. **22**(1), 49–71 (2009)
23. Crain, T., Gramoli, V., Larrea, M., Raynal, M.: DBFT: efficient leaderless Byzantine consensus and its application to blockchains. In: Symposium on Network Computing and Applications (NCA) (2018)
24. Dolev, D., Halpern, J.Y., Simons, B., Strong, R.: Dynamic fault-tolerant clock synchronization. J. ACM **42**(1), 143–185 (1995)
25. Dragoi, C., Widder, J., Zufferey, D.: Programming at the edge of synchrony. Proc. ACM Program. Lang. **4**(OOPSLA), 213:1-213:30 (2020)
26. Dutta, P., Guerraoui, R., Lamport, L.: How fast can eventual synchrony lead to consensus? In: Conference on Dependable Systems and Networks (DSN) (2005)
27. Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
28. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
29. Gafni, E.: Round-by-round fault detectors: unifying synchrony and asynchrony. In: Symposium on Principles of Distributed Computing (PODC) (1998)
30. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: scaling Byzantine agreements for cryptocurrencies. In: Symposium on Operating Systems Principles (SOSP) (2017)
31. Golan-Gueta, G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M.K., Seredinschi, D., Tamir, O., Tomescu, A.: SBFT: a scalable and decentralized trust infrastructure. In: Conference on Dependable Systems and Networks (DSN) (2019)
32. Haeberlen, A., Kuznetsov, P.: The fault detection problem. In: Conference on Principles of Distributed Systems (OPODIS) (2009)
33. Herzberg, A., Kutten, S.: Fast isolation of arbitrary forwarding faults. In: Symposium on Principles of Distributed Computing (PODC) (1989)
34. Herzberg, A., Kutten, S.: Early detection of message forwarding faults. SIAM J. Comput. **30**(4), 1169–1196 (2000)
35. Incorrect by construction-CBC Casper isn't live. https://derekhsorensen.com/docs/CBC_Casper_Flaw.pdf
36. Keidar, I., Shraer, A.: Timeliness, failure-detectors, and consensus performance. In: Symposium on Principles of Distributed Computing (PODC) (2006)
37. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
38. Malkhi, D., Reiter, M.: Unreliable intrusion detection in distributed computations. In: Workshop on Computer Security Foundations (CSFW) (1997)
39. Milosevic, Z., Biely, M., Schiper, A.: Bounded delay in Byzantine-tolerant state machine replication. In: Symposium on Reliable Distributed Systems (SRDS) (2013)
40. Mostéfaoui, A., Raynal, M.: Solving consensus using Chandra-Toueg's unreliable failure detectors: a general quorum-based approach. In: Symposium on Distributed Computing (DISC) (1999)
41. Naor, O., Baudet, M., Malkhi, D., Spiegelman, A.: Cogsworth: Byzantine view synchronization. In: Cryptoeconomics Systems Conference (CES) (2020)
42. Naor, O., Keidar, I.: Expected linear round synchronization: the missing link for linear Byzantine SMR. In: Symposium on Distributed Computing (DISC) (2020)
43. Simons, B., Welch, J., Lynch, N.: An overview of clock synchronization. In: Fault-Tolerant Distributed Computing (1986)
44. State machine replication in the Diem blockchain. https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf
45. Yin, M., Malkhi, D., Reiter, M. K., Golan-Gueta, G., Abraham, I.: HotStuff: BFT consensus with linearity and responsiveness. In: Symposium on Principles of Distributed Computing (PODC) (2019)