Live SMR in partitionable networks

Alejandro Naser Pastoriza
IMDEA Software Institute, Spain

Gregory Chockler
University of Surrey, UK

Alexey Gotsman
IMDEA Software Institute, Spain

Abstract

SMR protocols ensure the consistency of replicated state in spite of the failure of a fraction of its processes. Unfortunately, no deterministic algorithm implements SMR in an asynchronous network, even if a single process may crash. Therefore, these protocols often guarantee safety at all times and liveness only under synchronous periods. Following these results, there was an interest in understanding the degree of reliability and synchrony required to solve problems such as consensus or SMR. Apart from the theoretical interest, such weakenings of partial synchrony also have practical relevance: network partitions happen in practice and were a source of failures and outages.

We show that in a system with n processes and up to $f < n/2$ process crashes, SMR can be solved provided there exist some unknown correct process with f bidirectional links to correct processes that are eventually timely. In particular, the remaining $O(n^2)$ links may have transient or permanent failures. To this end, we extend the *SMR synchronizer* abstraction by Bravo et al. [6] to partitionable networks. We present a formal specification, its bounded-space implementation and use it to design a provably live SMR protocol. We show that these results are in a sense optimal: even if correct processes are eventually fully connected by timely links except for one whose either incoming or outgoing links may be faulty, then consensus cannot be solved.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases State machine replication, consensus, partitionable networks.

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

1 Introduction

SMR protocols ensure the consistency of replicated state in spite of the failure of a fraction of its processes. Unfortunately, no deterministic algorithm implements SMR in an asynchronous network, even if a single process may crash [8]. Therefore, these protocols often guarantee safety at all times and liveness only under synchronous periods. This is formalized by the *partial synchrony* model [7], which stipulates that after some unknown time the system becomes synchronous, with message delays bounded by an unknown constant and process clocks tracking real time.

Following these results, there was an interest in understanding the degree of reliability and synchrony required to solve consensus, and to implement abstractions such as Ω or failure detectors [1, 2]. Apart from the theoretical interest, such weakenings of partial synchrony also have practical relevance: network partitions happen in practice and were a source of failures and outages [3, 4, 9].

Guaranteeing liveness under partial synchrony is already nontrivial. The problem becomes even harder in the presence of transient link failures due to the difficulty to differentiate a transient link fault from a process crash. We show that in a system with n processes and up to $f < n/2$ process crashes, SMR can be solved provided there exist some unknown correct process with f bidirectional links to correct processes that are eventually timely. In particular, the remaining $O(n^2)$ links may have transient or permanent failures.

We follow an approach similar to that of [7] where protocols divide their execution into *views*, each with a designated leader process that coordinates the protocol execution. If the leader is faulty or does not have enough connectivity, the processes switch to another view with a different leader. To ensure liveness, an SMR protocol needs to spend sufficient time in views that are entered by sufficient correct processes and where the leader has enough connectivity. The challenge of achieving such *view synchronization* is that clocks can diverge and messages that could be used to synchronize processes can get lost or delayed. Even during the synchrony period faulty links between correct processes may selectively drop or delay messages, distorting the view of the system a process may have. *View synchronizers* [6, 5, 10, 11] encapsulate mechanisms for dealing with this challenge, allowing them to be reused across protocols. In this paper, we make the following contributions:

- We propose a formal specification that extends the *SMR synchronizer* abstraction from Bravo et al. [6] to the model of partitionable networks.
- We give a bounded-space implementation of an SMR synchronizer and prove that it satisfies our specification.
- We demonstrate the usability of the abstraction by constructing a provably live SMR protocol.
- We show that these results are in a sense optimal: even if correct processes are eventually fully connected by timely links except for one whose either incoming or outgoing links may be faulty, then consensus cannot be solved.

2 Related Work

In [4], the authors present a comprehensive study of system failures attributed to network partitioning faults in widely used distributed systems, and introduce a testing framework that can inject different types of network partitioning faults. In [3], the analysis is extended to other popular systems and they introduce a communication layer that is capable of masking partial network partitionings.

In [9], the authors study the Cloudflare outage on 2020-11-02, analyze Raft's behavior during partial network failures and introduce a benchmarking tool that allows to emulate a range of topologies and failures. Finally, they highlight that the `PreVote` optimisation can help with a particular type of failure where faulty nodes repeatedly call leader elections under partial network partitionings.

In [1], the authors study the feasability of implementing Ω in systems with weak reliability and synchrony assumptions. It is assumed that there is a timely correct process whose output links are eventually timely. However, since there are no guarantees on the connectivity of the output of Ω, this makes it unusable to solve consensus. In [2], these results are extended to study the degree of synchrony required to implement the leader election failure detector and solve consensus in partially synchronous systems. They show that to implement Ω and solve consensus, it is sufficient to have an unknown process having $f < n/2$ links that are eventually timely. However, it is assumed that the rest of the links are fair-lossy, making it an unsuitable model for network partitions.

It is worth noting that a solution to the problem of consensus in partitionable networks with transient link failures cannot rely in abstractions such as Ω or leader detectors. In fact, links could selectively let through messages sent by the underlying abstraction while dropping every message sent by the protocol atop it. As a consequence, the abstraction would be unable to identify that there is a problem, while rendering the protocol unusable.

This motivates the fact that such an underlying abstraction must accept hints from the protocol atop it to identify potentially faulty processes or links.

3 System Model

We assume a system $\mathcal{S}$ composed of $n = 2f + 1$ processes, out of which at most f can fail by crashing, i.e., permanently stopping execution. In the latter case the process is *faulty*, otherwise it is *correct*. Processes communicate through directed links that can lose, delay, reorder or duplicate messages, but not corrupt them. We also assume that processes are equipped with hardware clocks that can drift unboundedly from real time.

We consider a weakened *partial synchrony* model [7]. We say that a set $\mathcal{H}$ of at least $f+1$ correct processes is a *hub* if there exist a process $p \in \mathcal{H}$, called its *center*, and an unknown time $\mathsf{GST}_{\mathcal{H}}$ after which the following conditions hold: (i) for every $q \in \mathcal{H}$, message delays on the links between p and q are bounded by an unknown $\delta_{\mathcal{H}}$, and (ii) clocks at processes in $\mathcal{H}$ advance at the same rate as real time. We assume that in the system there exists at least one hub and let $\mathsf{GST} = \max_{\mathcal{H} \in \mathcal{S}} \mathsf{GST}_{\mathcal{H}}$ and $\delta = \max_{\mathcal{H} \in \mathcal{S}} \delta_{\mathcal{H}}$. Since there can exist only finitely many hubs, GST and δ are well defined.

4 SMR Synchronizer

We consider the synchronizer interface defined in [6, 10, 11]. Let the *views* be the set of the natural numbers, ranged over by v. The abstraction provides two primitives. A $\mathsf{new\_view}(v)$ notification at a process indicates that it must *enter* view v. An $\mathsf{advance}()$ request, allows a process to signal its wish to *advance* to a higher view.

This section relies on the following notation: given a view v entered by a process p_i, we denote by $E_i(v)$ the time when this happens. We let $E^{\mathcal{H}}_{\text{first}}(v)$ and $E^{\mathcal{H}}_{\text{last}}(v)$ denote respectively the earliest and the latest time when a process from a hub $\mathcal{H}$ enters v. Also, we let $E_{\text{first}}(v)$ and $E_{\text{last}}(v)$ denote respectively the earliest and the latest time when any process enters v. Given a view v from which a process p_i attempts to advance, we denote by $A_i(v)$ the time when this happens. We let $A^{\mathcal{H}}_{\text{first}}(v)$ and $A^{\mathcal{H}}_{\text{last}}(v)$ denote respectively the earliest and the latest time when a process from a hub $\mathcal{H}$ attempts to advance from v. Also, we let $A_{\text{first}}(v)$ and $A_{\text{last}}(v)$ denote respectively the earliest and the latest time when any process attempts to advance from v. Finally, given a partial function f, we write $f(x)\downarrow$ if $f(x)$ is defined, and $f(x)\uparrow$ if $f(x)$ is undefined.

4.1 Specification

Our first contribution is the extension of the SMR synchronizer specification to the model of partitionable networks, in Figure 1. Let $\mathcal{H}$ be an arbitrary hub.

The *Monotonicity* property ensures that, at any given process, its view can only increase. The *Validity* property ensures that a process may only enter view $v+1$ if some process in $\mathcal{H}$ has called $\mathsf{advance}$ while in v. This prevents a process p outside $\mathcal{H}$ from disrupting $\mathcal{H}$ by forcing view changes, in cases where p mistakenly suspects $\mathcal{H}$'s leader due to a faulty link. The *Bounded Entry* property ensures that, if some process from $\mathcal{H}$ enters a view v, then all processes from $\mathcal{H}$ will do so at most 2δ units of time of each other. This only holds if within 2δ no process from $\mathcal{H}$ attempts to advance to a higher view, as this may make some processes from $\mathcal{H}$ skip v and enter a higher view directly. Bounded Entry holds only starting from some view $\mathcal{V}$, since a synchronizer may not be able to guarantee it for views entered before GST. The *Startup* property ensures that if $f+1$ processes from $\mathcal{H}$ attempt to advance from view 0, then some process from $\mathcal{H}$ enters view 1. The *Progress* property determines the conditions under which some process from $\mathcal{H}$ will enter the next view $v+1$.

- **Monotonicity.** $\forall i, v, v'.\ E_i(v)\downarrow\ \wedge\ E_i(v')\downarrow\ \implies (v < v' \iff E_i(v) < E_i(v'))$
- **Validity.** $\forall i, v.\ E_i(v+1)\downarrow\ \implies A^{\mathcal{H}}_{\text{first}}(v)\downarrow\ \wedge\ A^{\mathcal{H}}_{\text{first}}(v) < E_i(v+1)$
- **Bounded Entry.** $\exists \mathcal{V}.\ \forall v \geq \mathcal{V}.\ E^{\mathcal{H}}_{\text{first}}(v)\downarrow\ \wedge \neg(A^{\mathcal{H}}_{\text{first}}(v) < E^{\mathcal{H}}_{\text{first}}(v) + 2\delta)$ $\implies (\forall p_i \in \mathcal{H}.\ E_i(v)\downarrow) \wedge (E^{\mathcal{H}}_{\text{last}}(v) \leq E^{\mathcal{H}}_{\text{first}}(v) + 2\delta)$
- **Startup.** $(\exists P \subseteq \mathcal{H}.\ |P| = f\ +\ 1\ \wedge\ (\forall p_i \in P.\ A_i(0)\downarrow)) \implies E^{\mathcal{H}}_{\text{first}}(1)\downarrow$
- **Progress.** $\forall v.\ E^{\mathcal{H}}_{\text{first}}(v)\downarrow\ \wedge\ (\exists P \subseteq \mathcal{H}.\ |P| = f+1\ \wedge\ (\forall p_i \in P.\ E_i(v)\downarrow \implies A_i(v)\downarrow))$ $\implies E^{\mathcal{H}}_{\text{first}}(v+1)\downarrow$

Figure 1 SMR synchronizer specification. The properties hold for every hub $\mathcal{H}$.

Because a process may mistakenly suspect a leader due to a faulty link, a view change cannot be triggered based on a single attempt to advance from a view v. Therefore, to switch to view $v+1$, a process must collect sufficient evidence: at least a majority of processes must have attempted to advance from v, thereby ensuring that at least one process from each hub whishes to do so. However, a process may wish to advance to a higher view, but all its outgoing links may be faulty, making it impossible to disseminate its intention. Thus, it can only be guaranteed that a process from $\mathcal{H}$ will enter $v+1$, if for some set P of $f+1$ processes from $\mathcal{H}$, any process in P entering v eventually invokes advance while in v.

4.2 Implementation

Our second contribution is an algorithm that implements the specification in Figure 1 under our system model. The implementation reuses algorithmic techniques from the SMR synchronizer by Bravo et al. [6]. However, to support a different model, it requires a different algorithm, correctness proof and analysis.

```
1  function advance():
2      send WISH(view + 1) to all;
3      advanced ← TRUE;
4  periodically
5      send ENTER(view) to all;
6      if advanced then
7          send WISH(view + 1) to all;
8  when received ENTER($v$)
9      if $v$ > view then
10         enter_view($v$);
11 when received WISH($v$) from $p$
12     views[$p$] ← max(views[$p$], $v$);
13     $v'$ ← max{$v$ | $\exists p$. views[$p$] = $v$ ∧
              |{$q$ | views[$q$] ≥ $v$}| ≥ $f+1$};
14     if $v'$ > view then
15         enter_view($v'$);
16 function enter_view($v$):
17     view ← $v$;
18     advanced ← FALSE;
19     trigger new_view(view);
20     send ENTER(view) to all;
```

Figure 2 A bounded-space SMR synchronizer. The periodic handler fires every ρ units of time.

When a process invokes advance() (line 1), the synchronizer does not immediately move to the next view $v+1$, but disseminates a WISH($v+1$) message announcing its intention. To reduce the space complexity, a process tracks only the highest view received from each process (line 12), kept in an array views. The array is then used to compute the $(f+1)$st highest view in views (line 13). A process enters a new view once it accumulates a sufficient number of WISH messages, and thus has collected enough evidence supporting this.

More specifically, a process enters the view stored in the v' variable when it is greater than its current view (lines 14 and 15). Thus, a process enters a view v only if it receives $f + 1$ WISHes for views $\geq v$, and a process may be forced to switch views even if it did not call advance; the latter helps lagging processes to catch up. Upon entering a view v (line 16), a new_view notification is triggered at the process, indicating that it must enter v (line 19). Next, the process disseminates an ENTER(v) message announcing that is has collected enough evidence to switch to v (line 20). Therefore, upon the receipt of the message (line 8), a process can safely enter view v without the need to collect $f + 1$ WISHes for a view $\geq v$ by itself (line 10). In this way, a process p can learn that there is enough evidence to enter a view, even if some WISHes originate at processes that are not directly connected to p through correct links.

Finally, to deal with message loss before GST, a process retransmits the most up to date information it has every ρ units of time, according to its local clock (line 4). It first retransmits the highest view for which it has collected enough evidence (line 5). Then, if the process has called advance in the current view (tracked by the advanced flag), it retransmits the highest WISH it has sent (line 7).

4.3 Correctness

We now show that the algorithm in Figure 2 correctly implements the specification in Figure 1. Proofs ommitted in this section can be found in §A. Let $\mathcal{H}$ be an arbitrary hub and p_i.view(t) denote p_i's view at the time t.

▶ **Lemma 1.** *If a process p_i enters a view v, then there exists a process $p_j \in \mathcal{H}$, a view $v' \geq v$ and a time $t < E_i(v)$ such that p_j sends* WISH(v') *at t.*

▶ **Lemma 2.** *If a process p_i sends* WISH($v + 1$) *at a time t, then $A_i(v)\downarrow \wedge A_i(v) \leq t$.*

▶ **Lemma 3.** *Validity holds: $\forall i, v.\ E_i(v + 1)\downarrow \implies A^{\mathcal{H}}_{\text{first}}(v)\downarrow \wedge A^{\mathcal{H}}_{\text{first}}(v) < E_i(v + 1)$.*

Proof. Let i and v be such that $E_i(v + 1)\downarrow$. By Lemma 1, there exists a process $p_j \in \mathcal{H}$, a view $v' \geq v + 1$ and a time $t < E_i(v + 1)$ such that p_j sends WISH(v') at t. Hence, by lines 2 and 7, p_j.view(t) $= v' - 1 \geq v$.

Suppose p_j.view(t) $= v$. Then at t, p_j sends WISH($v+1$). By Lemma 2, $A_j(v)\downarrow \wedge A_j(v) \leq t < E_i(v + 1)$. Therefore, $A^{\mathcal{H}}_{\text{first}}(v)\downarrow \wedge A^{\mathcal{H}}_{\text{first}}(v) \leq A_j(v) < E_i(v + 1)$. Suppose p_j.view(t) $> v$. Let p_k be the first process to enter a view $> v$ at a time $t_k \leq t$. By Lemma 1, there exists a process $p_l \in \mathcal{H}$, a view $v_l \geq p_k$.view(t_k) $> v$ and a time $t_l < t_k$ such that p_l sends WISH(v_l) at t_l. Hence, by lines 2 and 7, p_l.view(t_l) $= v_l - 1 \geq v$. Since $t_l < t_k$ and t_k is the earliest time at which a process has a view $> v$, p_l.view(t_l) $\leq v$. Hence, $v_l - 1 = p_l$.view(t_l) $= v$ and thus $v_l = v + 1$. Therefore, p_l sends WISH($v + 1$) at t_l and, by Lemma 2, $A_l(v)\downarrow \wedge A_l(v) \leq t_l < t_k \leq t < E_i(v + 1)$. Therefore, $A^{\mathcal{H}}_{\text{first}}(v)\downarrow \wedge A^{\mathcal{H}}_{\text{first}}(v) \leq A_l(v) < E_i(v + 1)$. ◀

▶ **Lemma 4.** $\forall v, v'.\ 0 < v < v' \wedge E_{\text{first}}(v')\downarrow \implies E^{\mathcal{H}}_{\text{first}}(v)\downarrow \wedge E^{\mathcal{H}}_{\text{first}}(v) < E_{\text{first}}(v')$.

Proof. Fix $v' \geq 2$ and assume that a process enters v', so that $E_{\text{first}}(v')\downarrow$. We prove by induction that for each k satisfying $1 \leq k \leq v' - 1$, some process in $\mathcal{H}$ enters $v' - k$ earlier than $E_{\text{first}}(v')$ and thus $E^{\mathcal{H}}_{\text{first}}(v' - k)\downarrow \wedge E^{\mathcal{H}}_{\text{first}}(v' - k) < E_{\text{first}}(v')$.

For the base case, assume that a process enters v'. Then by Lemma 3, there exists a process $p_i \in \mathcal{H}$ such that $A_i(v'-1)\downarrow \wedge A_i(v'-1) < E_{\text{first}}(v')$. Then p_i.view($A_i(v'-1)$) $= v'-1$ and thus $E_i(v' - 1)\downarrow \wedge E_i(v' - 1) \leq A_i(v' - 1) < E_{\text{first}}(v')$.

For the inductive step, assume that the required holds for some k so that $E^{\mathcal{H}}_{\text{first}}(v'-k)\downarrow \wedge E^{\mathcal{H}}_{\text{first}}(v'-k) < E_{\text{first}}(v')$. Then by Lemma 3, there exists a process $p_i \in \mathcal{H}$ such that $A_i(v'-k-1)\downarrow \wedge A_i(v'-k-1) < E^{\mathcal{H}}_{\text{first}}(v'-k)$. Then $p_i.\mathsf{view}(A_i(v'-k-1)) = v'-k-1$ and thus $E_i(v'-k-1)\downarrow \wedge E_i(v'-k-1) \le A_i(v'-k-1) < E^{\mathcal{H}}_{\text{first}}(v'-k) < E_{\text{first}}(v')$. ◀

▶ **Lemma 5.** *Consider a view $v > 0$ and assume that v is entered by some process in $\mathcal{H}$. If $E^{\mathcal{H}}_{\text{first}}(v) \ge \mathsf{GST}$ and no process in $\mathcal{H}$ attempts to advance from v before $E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$, then all processes in $\mathcal{H}$ enter v and $E^{\mathcal{H}}_{\text{last}}(v) \le E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$.*

Proof. Suppose there exist a process $p_j \in \mathcal{H}$ and a time $t \le E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$ such that $p_j.\mathsf{view}(t) = v' > v$. By Lemma 4, $E^{\mathcal{H}}_{\text{first}}(v+1)\downarrow \wedge E^{\mathcal{H}}_{\text{first}}(v+1) \le E^{\mathcal{H}}_{\text{first}}(v') \le t$. Thus, by Lemma 3, $A^{\mathcal{H}}_{\text{first}}(v)\downarrow \wedge A^{\mathcal{H}}_{\text{first}}(v) < E^{\mathcal{H}}_{\text{first}}(v+1) \le t \le E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$, contradicting the assumption that no process in $\mathcal{H}$ attempts to advance from v before $E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$. Therefore, for all times $t \le E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$ and processes $p_j \in \mathcal{H}$, $p_j.\mathsf{view}(t) \le v$.

Let p_c be the center of $\mathcal{H}$ and p_i be the process in $\mathcal{H}$ to enter v at the time $E^{\mathcal{H}}_{\text{first}}(v)$. Upon entering v, p_i sends an $\texttt{ENTER}(v)$ message to every process. Since the link between p_i and p_c is reliable after GST, the $\texttt{ENTER}(v)$ message is received by p_c no later than $E^{\mathcal{H}}_{\text{first}}(v) + \delta$, and thus p_c is guaranteed to enter v no later than $E^{\mathcal{H}}_{\text{first}}(v) + \delta$. Upon entering v, p_c sends an $\texttt{ENTER}(v)$ message to every process. Since the links between p_c and each process in $\mathcal{H}$ are reliable after GST, every process in $\mathcal{H}$ receives $\texttt{ENTER}(v)$ no later than $E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$.

Fix an arbitrary process $p_k \in \mathcal{H}$ and suppose $t_k \le E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$ is a time such that p_k receives the $\texttt{ENTER}(v)$ message sent by p_c at t_k. If $p_k.\mathsf{view}(t_k) = v$, then $E_k(v) < t_k \le E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$. If $p_k.\mathsf{view}(t_k) < v$, then line 9 guarantees that p_k enters v at t_k, and thus $E_k(v) = t_k \le E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$. Hence, $E^{\mathcal{H}}_{\text{last}}(v) \le E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$. ◀

▶ **Lemma 6.** *For all views $v, v' > 0$, if a process sends $\texttt{WISH}(v)$ before sending $\texttt{WISH}(v')$, then $v \le v'$.*

▶ **Lemma 7.** *For all processes $p_i \in \mathcal{H}$, times $t \ge \mathsf{GST} + \rho$, and views v, if p_i sends $\texttt{WISH}(v)$ at a time $\le t$, then there exists a view $v' \ge v$ and a time t' such that $\mathsf{GST} \le t' \le t$ and p_i either sends $\texttt{ENTER}(v')$ or $\texttt{WISH}(v')$ at t.*

▶ **Lemma 8.** *Startup holds: suppose there exists a set $P \subseteq \mathcal{H}$ of $f+1$ processes such that $\forall p_i \in P.\ A_i(0)\downarrow$. Then eventually some process in $\mathcal{H}$ enters view 1.*

Proof. Assume by contradiction that there exists a set $P \subseteq \mathcal{H}$ of $f+1$ processes such that $\forall p_i \in P.\ A_i(0)\downarrow$, and no process in $\mathcal{H}$ enters view 1. By Lemma 4, the latter implies

$$\forall v > 0.\ \forall p_i \in \mathcal{H}.\ E_i(v)\uparrow. \tag{1}$$

Then by Lemma 2, we have

$$\forall p_i.\ \forall t.\ \forall v > 1.\ \neg(p_i \text{ sends } \texttt{WISH}(v) \text{ at } t \ \wedge\ p_i \in \mathcal{H}). \tag{2}$$

Let $T_1 = \max(\mathsf{GST} + \rho, A^{\mathcal{H}}_{\text{last}}(0))$. Since each $p_i \in P$ attempts to advance from view 0, each $p_i \in P$ sends $\texttt{WISH}(1)$ no later than T_1. Since $T_1 \ge \mathsf{GST} + \rho$, by Lemma 7, there exists a view $v_i \ge 1$ and a time t_i such that $\mathsf{GST} \le t_i \le T_1$ and p_i either sends $\texttt{ENTER}(v_i)$ or $\texttt{WISH}(v_i)$ at t_i. If any p_i sends $\texttt{ENTER}(v_i)$, then $E^{\mathcal{H}}_{\text{first}}(v_i)\downarrow$, contradicting (1). Therefore, each p_i sends $\texttt{WISH}(v_i)$ at t_i. By (2), $v_i = 1$. Let p_c be the center of $\mathcal{H}$. Since the links between p_c and each p_i are reliable after GST, the $\texttt{WISH}(1)$ message sent by each p_i is received by p_c.

Thus, there exists a time $T_2 \ge T_1$ by which p_c has received the $\texttt{WISH}(1)$ from all processes in P. By (1), for all times t, $p_c.\mathsf{view}(t) = 0$. Also, all entries in $p_c.\mathsf{views}(T_2)$ associated with the processes in P are equal to 1. Since $|P| = f+1$: (i) at least $f+1$ entries in $p_c.\mathsf{views}(T_2)$ are equal to 1, and (ii) one of the $f+1$ highest entries in $p_c.\mathsf{views}(T_2)$ is equal to 1. From (i), $p_c.v'(T_2) \geq 1$, and from (ii), $p_c.v'(T_2) \leq 1$. Hence, $p_c.v'(T_2) = 1$, and therefore p_c enters view 1 by T_2, contradicting (1). ◀

▶ **Lemma 9.** *Progress holds: let $v > 0$ be a view such that $E^{\mathcal{H}}_{\text{first}}(v)\downarrow$ and $P \subseteq \mathcal{H}$ be such that $|P| = f+1$ and*

$$\forall p_i \in P.\ E_i(v)\downarrow \implies A_i(v)\downarrow. \tag{3}$$

Then $E^{\mathcal{H}}_{\text{first}}(v+1)\downarrow$.

Proof. Assume by contradiction that the required does not hold. Then there exists a view $v > 0$ such that some process in $\mathcal{H}$ enters v, (3) holds, and no process in $\mathcal{H}$ enters $v+1$. By Lemma, 4, the latter implies

$$\forall v' > v.\ \forall p_i \in \mathcal{H}.\ E_i(v')\uparrow. \tag{4}$$

Then by Lemma 2, we have

$$\forall p_i.\ \forall t.\ \forall v' > v+1.\ \neg(p_i \text{ sends } \mathtt{WISH}(v') \text{ at } t\ \wedge\ p_i \in \mathcal{H}). \tag{5}$$

Let $T_1 = \max(\mathsf{GST} + \rho, E^{\mathcal{H}}_{\text{first}}(v))$ and let $p_i \in \mathcal{H}$ be the first process to enter v. If $E^{\mathcal{H}}_{\text{first}}(v) \geq \mathsf{GST}$, then p_i sends an $\mathtt{ENTER}(v)$ message at GST or later. If $E^{\mathcal{H}}_{\text{first}}(v) < \mathsf{GST}$, and since after GST the p_i's local clock advances at the same rate as real time, there exists a time s satisfying $\mathsf{GST} \leq s \leq T_1$ such that p_i executes the retransmission code in line 4 at s. Therefore, at s, p_i sends an $\mathtt{ENTER}(v_i)$ message and since $p_i.\mathsf{view}$ is non-decreasing, $v_i \geq v$. If $v_i > v$, then $p_i.\mathsf{view}(s) > v$, contradicting (4). Therefore, $v_i \leq v$ and thus $v_i = v$. We conclude that in both cases p_i sends an $\mathtt{ENTER}(v)$ message at GST or later. Let p_c be the center of $\mathcal{H}$. Since the link between p_i and p_c is reliable after GST, p_c eventually receives the $\mathtt{ENTER}(v)$ message. By (4), $p_c.\mathsf{view} \leq v$ upon receipt of this message, and thus p_c eventually enters v. Upon entering v, p_c sends an $\mathtt{ENTER}(v)$ message, which, by the same argument, is eventually received by every process in $\mathcal{H}$. By (4), no process in $\mathcal{H}$ has $\mathsf{view} > v$ upon receipt of this message, and thus every process in $\mathcal{H}$ eventually enters v.

Consider an arbitrary process $p_j \in \mathcal{H}$ and suppose that p_j is a member of the set P stipulated by the lemma's premise. Since $E_j(v)\downarrow$, by (3), $A_j(v)\downarrow$. Let $T_2 = \max(\mathsf{GST} + \rho, A^{\mathcal{H}}_{\text{last}}(v))$. Then p_j has sent a $\mathtt{WISH}(v+1)$ by T_2. By Lemma 7, there exists a view $v_j \geq v+1$ and a time t' such that $\mathsf{GST} \leq t' \leq T_2$ and p_j either sends $\mathtt{ENTER}(v_j)$ or $\mathtt{WISH}(v_j)$ at t'. If p_j sends $\mathtt{ENTER}(v_j)$ then $p_j.\mathsf{view}(t') = v_j > v$, contradicting (4). Therefore, p_j sends $\mathtt{WISH}(v_j)$ at t'. By (5), $v_j \leq v+1$, and therefore $v_j = v+1$. Since the link between p_c and p_j is reliable after GST, the $\mathtt{WISH}(v+1)$ message sent by p_j is eventually received by p_c.

Thus, there exists a time $T_3 \geq T_2$ by which p_c has received the $\mathtt{WISH}(v+1)$ from all processes in P. By (4), for all times t, $p_c.\mathsf{view}(t) \leq v$. Also, all entries in $p_c.\mathsf{views}(T_3)$ associated with the processes in P are equal to $v+1$. Since $|P| = f+1$: (i) at least $f+1$ entries in $p_c.\mathsf{views}(T_3)$ are equal to $v+1$, and (ii), one of the $f+1$ highest entries in $p_c.\mathsf{views}(T_3)$ is equal to $v+1$. From (i), $p_c.v'(T_3) \geq v+1$, and from (ii), $p_c.v'(T_3) \leq v+1$. Hence, $p_c.v'(T_3) = v+1$, and therefore p_c enters view $v+1$ by T_2, contradicting (4). ◀

▶ **Theorem 10.** *The synchronizer satisfies its specification.*

Proof. Monotonicity is given by lines 9 and 14. Validity, Startup and Progress are given by Lemmas 3, 8 and 9, respectively. Let $\mathcal{V} = \max\{v \mid E^{\mathcal{H}}_{\text{first}}(v)\downarrow\ \wedge\ E^{\mathcal{H}}_{\text{first}}(v) < \mathsf{GST}\} + 1$. Then $\forall v \geq \mathcal{V}.\ E^{\mathcal{H}}_{\text{first}}(v)\downarrow \implies E^{\mathcal{H}}_{\text{first}}(v) \geq \mathsf{GST}$. Thus, by Lemma 5, Bounded Entry holds. ◀

Since the hub $\mathcal{H}$ was picked arbitrarily, properties in Figure 1 hold for every hub $\mathcal{H}$.

5 SMR in partitionable networks

Our third contribution is a demonstration of how the SMR synchronizer can be used to implement atomic broadcast in partitionable networks, from which SMR can be implemented in the standard way. The atomic broadcast abstraction provides two primitives. A broadcast(m) request, allows a process to send an application message to all other processes. A deliver(m) indication, allows a process to deliver a received message m to its application layer. We assume that all values broadcast in a single execution are unique.

Because in a partitionable network a correct process may have all its links be faulty, it is not possible to guarantee liveness for each of them. Hence, we only guarantee progress for correct processes with sufficient connectivity, i.e., processes in a hub. Thus, we say that an algorithm is a correct implementation of atomic broadcast in partitionable networks if its every execution satisfies:

Integrity. No message is delivered more than once.

Validity. If a process delivers m, then m was previously broadcast.

Agreement. If p_i delivers m_1 and p_j delivers m_2, then either p_i delivers m_2 or p_j delivers m_1.

Total order. If p_i delivers m_1 before m_2, then a process delivering m_2 also delivers m_1 before m_2.

Liveness. There exists a quorum $\mathcal{Q}$ such that: (i) if a process $p \in \mathcal{Q}$ broadcasts a message m, then p eventually delivers m, and (ii) if a message is delivered by some process, then m is eventually delivered by every $p \in \mathcal{Q}$.

5.1 Implementation

Overview. The protocol uses a leader-based approach. A leader wins an election if it receives votes from a majority, thereby obtaining the right to propose messages. Once elected, the leader has started a period of execution called a *view*. At any time, each process participates in a single view stored in the variable view.

The leader receives incoming messages and appends them at the end of the array log, thereby dictating the order in which they should be delivered. To this end, it tracks the last non-empty slot in the variable next. For fault tolerance, the ordering of the messages needs to be finalized through consensus, which stipulates a round of notifications from the followers to the leader. Therefore, the leader propagates the messages to the followers. Upon receiving a proposal, the followers store it in its local copy of the log array and notify its receiption to the leader, thereby accepting it. If a majority of followers accept the message, its survival to any tolerated failure is guaranteed and can therefore be delivered at the participating processes. The processes persist the position of the last delivered message in the log array in a variable last_delivered.

When a failure is suspected the processes execute a recovery protocol. The purpose of the recovery protocol is to elect a new leader that has convinced a quorum of processes to join its view, and to agree on a common consistent state that is at least as up to date as the state of any of the participating processes. The second requirement guarantees that the new leader preserves any messages delivered in previous views. A variable status tracks whether the process is a LEADER, a FOLLOWER or it is in a special RECOVERING state used during leader changes. To guarantee a consistent state, it is required that the followers synchronize their state with the leader before they start accepting messages. Since there can exist multiple failed election attempts, we use an additional view variable cview to represent the last view in which a process has synchronized its state with that of the leader.

```
1  periodically
2      pre: status = LEADER;
3      broadcast(nop);
4  when received broadcast($m$)
5      periodically until $m$ is delivered
6          send BROADCAST($m$) to leader(view);
7          if timer_delivery[$m$] is disabled then
8              start_timer(timer_delivery[$m$],
                   dur_delivery);
9  when received BROADCAST($m$)
10     pre: status = LEADER;
11     pre: $m = \mathbf{nop} \lor \forall k.\ \mathsf{log}[k] \neq m$;
12     log[next] ← $m$;
13     send ACCEPT(view, next, $m$) to all;
14     next ← next + 1;
15 when received ACCEPT($v, k, m$) from $p_j$
16     pre: view = $v$ ∧
         status ∈ {LEADER, FOLLOWER};
17     log[$k$] ← $m$;
18     send ACCEPT_ACK($v, k$) to $p_j$;
19 when received a quorum of
     ACCEPT_ACK($v, k$)
20     pre: view = $v$ ∧ status = LEADER;
21     send COMMIT($v, k$, log[$k$]) to all;
22 when received COMMIT($v, k, m$) from $p_j$
23     pre: view = $v$ ∧ last_delivered + 1 = $k$;
24     log[$k$] ← $m$;
25     last_delivered ← $k$;
26     if $m \neq$ nop then
27         deliver(log[$k$]);
28     stop_timer(timer_commit);
29     stop_timer(timer_delivery[$m$]);
30     start_timer(timer_commit, dur_commit);
```

Figure 3 The protocol: failure-free case. The periodic handler fires every ρ units of time.

Failure-free case. In the normal operation of the protocol, a leader proposes a sequence of messages to its followers to replicate them and ensure the durability of the delivered messages and its order. Its code is shown in Figure 3.

To transmit a message m, a process sends it in a BROADCAST message (line 6). It keeps sending the message until it is delivered by the process. The periodic retransmission ensures that the message will reach the processes despite message loss before GST.

A process acts on the BROADCAST message only when it is the leader (line 10). Upon receiving the message, the leader checks whether it has previously received it to avoid delivering the same message more than once (line 11). Otherwise, it appends the message to the ordered sequence log and performs an analogous to the Phase 2 of Paxos, trying to convince the processes to accept its proposal. To this end, it sends an ACCEPT message, that is the analogous to the 2A message of Paxos (line 13). The message carries the view, the position of the message in the log array and the message m.

A process acts on the ACCEPT message only if it participates in the corresponding view (line 16). It stores the message in its local copy of the log array and then sends an ACCEPT_ACK message to its leader, analogous to the 2B message of Paxos (line 18). The message carries the view and the slot number corresponding to the message being acknowledged.

The leader of the view acts once it receives ACCEPT_ACK messages for a slot k from each process in a quorum (line 19), in which case the message has been present in the array of a majority in the same view and slot, and therefore it can survive any tolerated failure. In this case, the leader notifies the followers that the message can be safely delivered through a COMMIT message (line 21). The message carries the view, the position of the message in the log array and the message m.

Upon receiving a COMMIT message (line 22), the process updates last_delivered to track the prefix of the common global sequence of messages that it has delivered and updates its array. The messages are delivered in increasing order of slot numbers without gaps as implied by line 23. Finally, the process delivers the message to its application layer (line 27).

We explain the timers and the broadcast of **nops** in the recovery case.

```
31 function start():
32     if view = 0 then
33         advance();

34 upon new_view($v$)
35     view ← $v$;
36     status ← RECOVERING;
37     send STATE($v$, cview, log) to leader($v$);
38     stop_all_timers();
39     start_timer(timer_recovery, dur_recovery);

40 when received STATE($v, cview_j, log_j$) from
     each $p_j$ in a quorum $\mathcal{Q}$
41     pre: view = $v$ ∧ status = RECOVERING;
42     let $j_0$ be such that
       $\forall p_j \in \mathcal{Q}.\ (cview_{j_0}, |log_{j_0}|) \geq (cview_j, |log_j|)$;
43     log ← $log_{j_0}$;
44     send NEW_STATE($v$, log) to $\mathcal{P} \setminus \{p_i\}$;

45 when received NEW_STATE($v, log$) from $p_j$
46     pre: view = $v$ ∧ status = RECOVERING;
47     log ← $log$;
48     cview ← $v$;
49     status ← FOLLOWER;
50     send NEW_STATE_ACK($v$) to $p_j$;
51     stop_timer(timer_recovery);
52     start_timer(timer_commit, dur_commit);

53 when received NEW_STATE_ACK($v$)
     from a set $Q$ such that $Q \cup \{p_i\}$ is a
     quorum
54     pre: view = $v$ ∧ status = RECOVERING;
55     cview ← $v$;
56     status ← LEADER;
57     next ← $\max\{k \mid \text{log}[k] \neq \bot\} + 1$;
58     for $\{k \mid \text{log}[k] \neq \bot\}$ do
59         send COMMIT($v, k$, log[$k$]) to all;
60     stop_timer(timer_recovery);
61     start_timer(timer_commit, dur_commit);

62 when a timer expires
63     stop_all_timers();
64     dur_commit ← dur_commit + $\tau$;
65     dur_delivery ← dur_delivery + $\tau$;
66     dur_recovery ← dur_recovery + $\tau$;
67     advance();
68     status ← ADVANCED;
```

Figure 4 The protocol: recovery case.

Recovery case. We next explain how the protocol deals with failures by executing the recovery protocol shown in Figure 4. The goal of the recovery protocol is two-fold: first, elect a leader who convinces a quorum of processes to participate in its view and to choose a state that *dominates* the quorum, i.e., that is at least as up to date as the one of any of the participating processes; and secondly, guarantee that before a follower starts accepting proposals from the leader, it has synchronized its state with that of the leader, ensuring this way that they remain in synchrony.

Triggering view changes. We now describe when a process calls advance, which is key to ensure liveness. This occurs either on start-up (line 33) or when the process suspects a failure. To this end, the process monitors that progress is being made in its current view using timers; if one of these expires, the process calls advance and sets its status to ADVANCED (lines 67 and 68). First, the process checks that each message it receives is delivered promptly to guard against leader failures. For a message m this is done using timer_delivery[m], set for a duration dur_delivery when the process receives broadcast(m) (line 8). To ensure that this timer is not repeatedly reset by the periodic handler in line 5 and thus not allowed to expire, timer_delivery[m] is set only if it is not already enabled. The timer is stopped when the process delivers m (line 29). Secondly, the process checks that the leader initializes its view quickly enough to guard against the leader crashing or advancing to a higher view during initialization. Thus, when a process enters a view it starts timer_recovery for a duration dur_recovery (line 39). The process stops the timer when it becomes either a FOLLOWER or a LEADER in the current view, and thus it has finished synchronizing its state with that of the leader. Thirdly, the process monitors the leader's behavior to check that it is correct, and that is is supported by a quorum of correct processes through sufficiently good links. It does so by checking that it often receives `COMMIT` messages from the leader, and thus the leader was able to receive a round of notifications from its followers. Thus, upon becoming a LEADER or a FOLLOWER the process starts timer_commit for a duration dur_commit (lines 52 and 61). The timer is stopped when the process receives a `COMMIT` message (line 28). However, to continue monitoring the correct behavior in the current view, the process restarts the timer (line 30). Notice that in particular, this allows the process to check that it delivers all messages in the initial log. Finally, since it may happen that the leader has no message to propose, it is possible that no `COMMIT` messages are sent. This is addressed by the leader by periodically sending **nop**s, which represent void operations and which are not delivered to the application layer (line 26).

The above checks may make a process mistakenly suspect a failure if the timeouts are initially set too small with respect to the unknown message delay δ. Thus, a process increases dur_commit, dur_delivery and dur_recovery each time a timer expires (lines 64, 65 and 66).

View initialization. When the synchronizer indicates to a process that it must enter a new view v (line 34), the process sets view to v, ensuring that it will no longer accept messages from prior views. It also sets status to RECOVERING, signaling that the process is not yet ready to order messages in the new view. It then sends a `STATE` message to the leader of v with the information about the last view in which it synchronized its state with that of the leader, and the log of messages it has accepted in that view. This is the analogous to the `1B` message of Paxos (line 37).

The leader acts once it receives `STATE` messages from a quorum of processes. It then picks the most up to date state as the initial log of the new view (line 42). To finalize the recovery procedure, the leader must synchronize its state with that of the followers. To this end, it sends a `NEW_STATE` message carrying the view and its initial log (line 44).

Upon receipt of a `NEW_STATE` message, a process overwrites its state with the one provided by the leader, sets its status to FOLLOWER an establishes its view number, registering this way that it has synchronized its state with that of the leader (lines 47, 48 and 49). It then notifies the reception of the new state by sending a `NEW_STATE_ACK` message (line 50).

Upon receipt of `NEW_STATE_ACK` messages from a set of processes that together with the leader form a quorum, the leader knows that a majority of processes share its state in its view and therefore sets its view number (line 55), establishes as its leader (line 56), sets next to the last non-empty slot in the log array (line 57) and notifies that all messages in its log can be safely delivered (line 58).

5.2 Correctness

We now prove that the protocol satisfies the Liveness property of atomic broadcast. Proofs ommitted in this section can be found in §B. We say that a process *predelivers* a message m if it executes lines 24-30 upon receipt of a $\texttt{COMMIT}(\_, \_, m)$ message. Fix an arbitrary hub $\mathcal{H}$ and let p_c be its center.

▶ **Lemma 11.** *Let p_i be a correct process in view v that never enters a view higher than v. If p_i predelivers only finitely many messages or it never predelivers a message that it has broadcast while in v, then it eventually calls* advance *in v.*

▶ **Lemma 12.** *Consider a view $v \geq \mathcal{V}$ such that $E^{\mathcal{H}}_{\text{first}}(v) \geq \mathsf{GST}$ and $\mathtt{leader}(v) = p_c$. Let $p_i \in \mathcal{H}$ be a process that enters v. If we have $\mathsf{dur\_recovery}_i(v) > 5\delta$, $\mathsf{dur\_delivery}_i(v) > 6\delta$ and $\mathsf{dur\_commit}_i(v) > \rho + 6\delta$, then p_i is not the first process in $\mathcal{H}$ to call* advance *in v.*

Proof. Since $E^{\mathcal{H}}_{\text{first}}(v) \geq \mathsf{GST}$, messages between processes in $\mathcal{H}$ and p_c sent after $E^{\mathcal{H}}_{\text{first}}(v)$ get delivered within δ and clocks at processes in $\mathcal{H}$ track real time. By contradiction, assume that p_i is the first process in $\mathcal{H}$ to call advance in v. This only occur if a timer expires at p_i.

Suppose that timer_recovery expires at p_i. Because p_i is the first process to call advance in v and $\mathsf{dur\_recovery}_i(v) > 5\delta$, no process in $\mathcal{H}$ calls advance in v until after $E^{\mathcal{H}}_{\text{first}}(v) + 5\delta$. Then by Bounded Entry all processes in $\mathcal{H}$ enter v by $E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$. By Validity no process can enter $v + 1$ until after $E^{\mathcal{H}}_{\text{first}}(v) + 5\delta$, and by Lemma 4 the same holds for any view $> v$. Thus, all processes in $\mathcal{H}$ stay in v at least until $E^{\mathcal{H}}_{\text{first}}(v) + 5\delta$.

When a process in $\mathcal{H}$ enters v it sends a STATE message to p_c, which happens by $E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$. When p_c receives STATE messages from a quorum of processes, it broadcasts a NEW_STATE message. Thus, by $E^{\mathcal{H}}_{\text{first}}(v) + 4\delta$ all processes in $\mathcal{H}$ other than p_c receive this message, thereby stopping the timer timer_recovery (line 51). Upon receipt of a NEW_STATE message, a process replies with a NEW_STATE_ACK (line 50). Thus, by $E^{\mathcal{H}}_{\text{first}}(v) + 5\delta$ the process p_c receives a quorum of NEW_STATE_ACK messages, thereby stopping the timer timer_recovery (line 60). In either case, p_i is guaranteed to stop the timer timer_recovery by $E^{\mathcal{H}}_{\text{first}}(v) + 5\delta$, which contradicts the assumption that timer_recovery expires at p_i while in v.

Suppose that timer_commit expires at p_i after being initiated at a time $t \geq E^{\mathcal{H}}_{\text{first}}(v)$. Because p_i is the first process in $\mathcal{H}$ to call advance in v and $\mathsf{dur\_commit}_i(v) > \rho + 6\delta$, no process in $\mathcal{H}$ calls advance in v until after $t + \rho + 6\delta$. Then, by Bounded Entry all processes in $\mathcal{H}$ enter v by $E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$. By Validity no process can enter $v + 1$ until after $t + \rho + 6\delta$, and by Lemma 4 the same holds for any view $> v$. Thus, all processes in $\mathcal{H}$ stay in v at least until $t + \rho + 6\delta$.

By the time t, the process p_c has sent a NEW_STATE message that is received by every other process in $\mathcal{H}$ no later than $t + \delta$. Since all processes in $\mathcal{H}$ enter v by $t + 2\delta$, every process in $\mathcal{H}$ that is $\neq p_c$ is guaranteed to be a FOLLOWER by this time. Thus, by $t + 3\delta$, p_c receives NEW_STATE_ACK from every other process in $\mathcal{H}$, thereby becoming a LEADER. Since the periodic handler at line 1 fires every ρ units of time and the clock at p_c tracks real time, it is guaranteed that p_c sends an $\mathtt{ACCEPT}(v, k, \mathbf{nop})$ message by $t + \rho + 3\delta$, with $k \geq |p_c.\mathsf{log}(t)| \geq |p_i.\mathsf{log}(t)| \geq p_i.\mathsf{last\_delivered}(t)$. Thus, by $t + \rho + 4\delta$, every process in $\mathcal{H}$ receives and replies this message with an $\mathtt{ACCEPT\_ACK}(v, k)$, received by p_c by $t + \rho + 5\delta$. Upon receipt of these messages, p_c sends a $\mathtt{COMMIT}(v, k, \mathbf{nop})$, received by every process in $\mathcal{H}$ by $t + \rho + 6\delta$. Thus, by $t + \rho + 6\delta$, p_i receives a $\mathtt{COMMIT}(v, k, \mathbf{nop})$ message with $k \geq p_i.\mathsf{last\_delivered}(t) + 1$. The fifoness of the links guarantees that before receiving $\mathtt{COMMIT}(v, k, \_)$, p_i received $\mathtt{COMMIT}(v, k', \_)$ for all k' satisfying $1 \leq k' \leq k$. Therefore, p_i has received a $\mathtt{COMMIT}(v, k', \_)$ message with $k' = p_i.\mathsf{last\_delivered}(t) + 1$ no later than $t + \rho + 6\delta$, thereby stopping the timer timer_commit (line 28), which contradicts the assumption that timer_commit expires at p_i while in v.

Suppose that timer_delivery$[m]$ expires at p_i after being initiated at a time $t \geq E^{\mathcal{H}}_{\text{first}}(v)$. Because p_i is the first process in $\mathcal{H}$ to call advance in v and $\mathsf{dur\_delivery}_i(v) > 6\delta$, no process in $\mathcal{H}$ calls advance in v until after $t + 6\delta$. Then, by Bounded Entry all processes in $\mathcal{H}$ enter v by $E^{\mathcal{H}}_{\text{first}}(v) + 2\delta$. By Validity no process can enter $v + 1$ until after $t + 6\delta$, and by Lemma 4 the same holds for any view $> v$. Thus, all processes in $\mathcal{H}$ stay in v at least until $t + 6\delta$.

By the time t, the process p_c has sent a NEW_STATE message that is received by every other process in $\mathcal{H}$ no later than $t + \delta$. Since all processes in $\mathcal{H}$ enter v by $t + 2\delta$, every process in $\mathcal{H}$ that is $\neq p_c$ is guaranteed to be a FOLLOWER by this time. Thus, by $t + 3\delta$, p_c receives

`NEW_STATE_ACK` from every other process in $\mathcal{H}$, thereby becoming a LEADER. Furthermore, by $t+3\delta$ the process p_c receives the `BROADCAST`(m) message sent by p_i at t (line 6). Then there exists k such that p_c sends an `ACCEPT`(v,k,m) message by $t+3\delta$. Thus, by $t+4\delta$, every process in $\mathcal{H}$ receives and replies this message with an `ACCEPT_ACK`(v,k), received by p_c by $t+5\delta$. Upon receipt of these messages, p_c sends a `COMMIT`(v,k,m), received by every process in $\mathcal{H}$ by $t+6\delta$. Thus, by $t+6\delta$, p_i receives a `COMMIT`(v,k,m). The fifoness of the links guarantees that before receiving `COMMIT`(v,k,m), p_i received `COMMIT`$(v,k',\_)$ for all k' satisfying $1 \le k' \le k$. Therefore, by $t+6\delta$, p_i has $\mathsf{last\_delivered}+1 = k$ and thus it stops the timer $\mathsf{timer\_delivery}[m]$ (line 29), which contradicts the assumption that $\mathsf{timer\_delivery}[m]$ expires at p_i while in v. ◀

▶ **Lemma 13.** *Consider a view $v \ge \mathcal{V}$ such that $E^{\mathcal{H}}_{\text{first}}(v) \ge \mathsf{GST}$ and $\mathtt{leader}(v) = p_c$. If at each process $p_i \in \mathcal{H}$ that enters v we have $\mathsf{dur\_recovery}_i(v) > 5\delta$, $\mathsf{dur\_delivery}_i(v) > 6\delta$ and $\mathsf{dur\_commit}_i(v) > \rho + 6\delta$, then no process in $\mathcal{H}$ calls $\mathsf{advance}$ in v.*

▶ **Lemma 14.** *Let v be a view and $\mathcal{Q}$ be a quorum of correct processes such that*

$$\exists t.\ \forall p_i \in Q.\ \forall t' \ge t.\ p_i.\mathsf{view}(t') = v \wedge p_i.\mathsf{status}(t') \in \{\textit{LEADER}, \textit{FOLLOWER}\}. \tag{6}$$

Then (i) if a process $p \in \mathcal{Q}$ broadcasts a message m, then p eventually delivers m, and (ii) if a message m is delivered by some process, then m is eventually delivered by every $p \in \mathcal{Q}$.

Proof. Suppose that a process $p_j \in \mathcal{Q}$ broadcasts a message $m \ne \textbf{nop}$ but that it never predelivers it. Since as long as p_j does not predeliver m it continues to periodically broadcast it (line 6), there exists a time at which p_j broadcasts m while in v. Since p_j never predelivers m and never enters a view higher than v, by Lemma 11, p_j calls $\mathsf{advance}$ while in v and sets its status to ADVANCED, contradicting (6). Therefore, p_j eventually predelivers m. Since $m \ne \textbf{nop}$, p_j delivers m.

Suppose now that a process delivers a message $m \ne \textbf{nop}$ upon receipt of a `COMMIT`$(\_,k,m)$ message. Let p_j be an arbitrary process in $\mathcal{Q}$. We show that $p_j.\mathsf{last\_delivered}$ is unbounded. By contradiction, assume there exists a k' such that $p_j.\mathsf{last\_delivered}(t) \le k'$ for all times t and $p_j.\mathsf{last\_delivered}(t') = k'$ for some t'. Then, the timer $\mathsf{timer\_commit}$ eventually expires at p_j while in view v. Indeed, upon becoming a LEADER or a FOLLOWER at v, or when it predelivers $p_j.\mathsf{log}[k']$, whichever happens last, the process p_j starts the timer $\mathsf{timer\_commit}$ (lines 52, 61 and 30). The precondition at line 23 ensures that this timer can only be stopped upon receipt of a `COMMIT`$(v,k'+1,\_)$ message. However, if this was the case, p_j would increase $p_j.\mathsf{last\_delivered}$ to $k'+1$ (line 25) contradicting the fact that it is bounded by k'. Therefore, $\mathsf{timer\_commit}$ eventually expires and thus p_j calls $\mathsf{advance}$ while in v. As a consequence, p_j sets its status to ADVANCED, contradicting (6). Hence, there exists a time t' such that $p_j.\mathsf{last\_delivered}(t') \ge k$. Therefore, there exists a time $t'' \le t'$ such that p_j receives `COMMIT`$(\_,k,m')$ at t'' upon which p_j predelivers m'. By safety $m' = m$ and since $m \ne \textbf{nop}$, p_j delivers m. Since p_j was picked arbitrarily, every process in $\mathcal{Q}$ delivers m. ◀

▶ **Lemma 15.** *There exists a quorum $\mathcal{Q}$ such that: (i) if a process $p \in \mathcal{Q}$ broadcasts a message m, then p eventually delivers m, and (ii) if a message m is delivered by some process, then m is eventually delivered by every $p \in \mathcal{Q}$.*

Proof. By contradiction, assume there is no quorum validating the lemma. We prove that in this case the protocol keeps moving through views forever.

Claim 1. Every view is entered by some process in $\mathcal{H}$.

Proof. Since all processes in $\mathcal{H}$ call start, by Startup a process in $\mathcal{H}$ eventually enters view 1. Assume the contrary, so that there exists a maximal view v entered by any process in $\mathcal{H}$. Let $P \subseteq \mathcal{H}$ be any set of $f+1$ processes and consider an arbitrary process $p_i \in P$ that enters v.

We show that p_i must predeliver only finitely many messages while in view v. Assume the contrary, so that p_i predelivers infinitely many messages while in view v. The precondition at line 23 guarantees that the process does so upon receipt of $\texttt{COMMIT}(v, k', \_)$ for all $k' \geq k$, for some k. Therefore, for each k', there exists a quorum of processes $\mathcal{Q}_{k'}$ that reply to the $\texttt{ACCEPT}(v, k', \_)$ message sent by $\texttt{leader}(v)$ with an $\texttt{ACCEPT\_ACK}(v, k')$. Since there exist only finitely many quorums, there exists a quorum $\mathcal{Q}$ of correct processes such that $\mathcal{Q} = \mathcal{Q}_{k'}$ for infinitely many values of k'. The precondition at line 16 and the fact that view never decreases at a process ensures that

$$\exists t.\ \forall p_i \in Q.\ \forall t' \geq t.\ p_i.\mathsf{view}(t') = v \wedge p_i.\mathsf{status}(t') \in \{\text{LEADER}, \text{FOLLOWER}\} \tag{7}$$

Thus, by Lemma 14, $\mathcal{Q}$ validates the lemma, contradicting the hypothesis that there is no such quorum. Therefore, it cannot be the case that p_i predelivers infinitely many messages while in view v. Then by Lemma 11, p_i eventually calls advance while in v. Since p_i was picked arbitrarily, we have $\forall p_i \in P.\ E_i(v)\downarrow \implies A_i(v)\downarrow$. By Progress, $E^{\mathcal{H}}_{\text{first}}(v+1)\downarrow$, which yields a contradiction. Thus, processes in $\mathcal{H}$ keep entering views forever. The claim follows from Lemma 4 ensuring that, if a view is entered by a process in $\mathcal{H}$, then so are all preceding views. ◀

Let view v_1 be the first view such that $v_1 \geq \mathcal{V}$ and $E^{\mathcal{H}}_{\text{first}}(v_1) \geq \mathsf{GST}$; such a view exists by Claim 1. We next show that processes in $\mathcal{H}$ will increase their timeouts high enough to satisfy the bounds in Lemma 13.

Claim 2. Every process in $\mathcal{H}$ calls the timer expiration handler (line 62) infinitely often.

Proof. Assume the contrary and let C_{fin} and C_{inf} be the sets of processes in $\mathcal{H}$ that call the timer expiration handler finitely and infinitely often, respectively. Then $C_{\text{fin}} \neq \varnothing$, and by Claim 1 and Validity, $C_{\text{inf}} \neq \varnothing$. The values of dur_commit, dur_delivery and dur_recovery increase unboundedly at processes from C_{inf}, and do not change after some view v_2 at processes from C_{fin}. By Claim 1 and since leaders rotate round-robin, there exists a view $v_3 \geq \max\{v_1, v_2\}$ led by p_c such that any process $p_i \in C_{\text{inf}}$ that enters v_3 has $\mathsf{dur\_commit}_i > \rho + 6\delta$, $\mathsf{dur\_delivery}_i > 6\delta$ and $\mathsf{dur\_recovery}_i > 5\delta$. By Claim 1 and Validity, at least one process in $\mathcal{H}$ calls advance in v_3; let p_l be the first process to do so. Because $v_3 \geq v_2$, this process cannot be in C_{fin}, since none of these processes can increase their timers in v_3. Then $p_l \in C_{\text{inf}}$, contradicting Lemma 13. ◀

By Claims 1 and 2, there exists a view $v_4 \geq v_1$ led by p_c such that some process in $\mathcal{H}$ enters v_4, and for any process $p_i \in \mathcal{H}$ that enters v_4 we have $\mathsf{dur\_commit}_i > \rho + 6\delta$, $\mathsf{dur\_delivery}_i > 6\delta$ and $\mathsf{dur\_recovery}_i > 5\delta$. By Lemma 13, no process in $\mathcal{H}$ calls advance in v_4. On the other hand, by Claim 1, some process in $\mathcal{H}$ enters $v_4 + 1$. By Validity, some process in $\mathcal{H}$ calls advance in v_4, which is a contradiction. ◀

6 An impossibility result

Our fourth contribution is to show that our results are in a sense optimal: even if correct processes are eventually fully connected by timely links except for one whose either incoming or outgoing links may be faulty, then consensus cannot be solved.

An impossibility result for an f-clique + 1 incoming link. Assume a model where there is a time after which there are f correct processes fully connected by timely links and one extra incoming timely link into the clique from an $(f+1)$st correct process.

▶ **Theorem 16.** *There does not exist a deterministic algorithm implementing consensus in the above model.*

Proof. Suppose for the sake of contradiction that there exists such an algorithm. Let G_1 be the set $\{1, \ldots, f\}$, G_2 be the set $\{f+1\}$ and G_3 be the set $\{f+2, \ldots, n\}$.

Let C_1 be a configuration where $G_1 \cup G_2$ form the majority stipulated by the system model where processes from G_1 constitute a clique and there is a timely link from process $f+1$ into G_1. Furthermore, assume that processes from G_3 are initially crashed and that any other link from/to process $f+1$ drop all messages. Let α_1 be an execution of the system in C_1 that contains an invocation $\mathit{propose}(v_1)$ by process 1 and no other invocations. By Termination, a process from G_1 eventually decides with a matching $\mathit{decide}(v_1)$. Let α_1' be the prefix of α_1 ending with $\mathit{decide}(v_1)$.

Let C_2 be a configuration where $G_3 \cup G_2$ form the majority stipulated by the system model where processes from G_3 constitute a clique and there is a timely link from process $f+1$ into G_3. Furthermore, assume that processes from G_1 are initially crashed and that any other link from/to process $f+1$ drop all messages. Let α_2 be an execution of the system in C_2 that contains an invocation $\mathit{propose}(v_2)$ by process n with $v_2 \neq v_1$ and no other invocations. By Termination, a process from G_3 eventually decides with a matching $\mathit{decide}(v_2)$. Let α_2' be the prefix of α_2 ending with $\mathit{decide}(v_2)$.

Let s_1 and s_2 be the sequences of actions performed by process $f+1$ in α_1' and α_2', respectively. Since the process $f+1$ is deterministic, either s_1 is a prefix of s_2 or s_2 is a prefix of s_1. Let $\beta = s_2 - s_1$ if s_1 is a prefix of s_2 or ϵ otherwise. Let $\alpha_1'' = \alpha_1'\beta$ be an execution in which all actions occur before GST. Since the actions performed by the process $f+1$ do not depend on any input, this is a valid execution in C_1. Let $\gamma = s_1 - s_2$ if s_2 is a prefix of s_1 or ϵ otherwise. Let $\alpha_2'' = \alpha_2'\gamma$ be an execution in which all actions occur before GST. Since the actions performed by the process $f+1$ do not depend on any input, this is a valid execution in C_2. Notice that α_1'' and α_2'' satisfy $\alpha_1''|_{f+1} = \alpha_2''|_{f+1}$.

Let C_3 be a configuration where G_1 and G_3 form cliques made out of timely links. Also, assume that there is a timely link from process $f+1$ into G_1 and a timely link from process $f+1$ into G_3. Finally assume that every link between G_1 and G_3 as well as any other link from/to process $f+1$ drop all messages. Note that this is allowed by the system model. Let α be an execution of the system in C_3 that begins with all the activity from α_1'' followed by all the activity from α_2'', except for all the actions by process $f+1$ in α_2''. Also, assume that all actions in α occur before GST. Notice that if any process in G_3 delivers a message sent by the process $f+1$ in α_2'', and thus in α, then the message has been previously sent in α. Thus, this is a valid execution. Furthermore, for any process $p_i \in G_1 \cup G_2$, $\alpha|_{p_i} = \alpha_1''|_{p_i}$ and thus α is indistinguishable from α_1'' to the processes in $G_1 \cup G_2$. Similarly, for any process $p_i \in G_3 \cup G_2$, $\alpha|_{p_i} = \alpha_2''|_{p_i}$ and thus α is indistinguishable from α_2'' to the processes in $G_3 \cup G_2$. Finally, since $\mathit{decide}(v_1)$ occurs in α_1'' and $\mathit{decide}(v_2)$ occurs in α_2'', then both $\mathit{decide}(v_1)$ and $\mathit{decide}(v_2)$ occur in α. This violates the Agreement property, thus yielding a contradiction. ◀

In Appendix §C we show that there is no deterministic algorithm implementing consensus for the case of an f-clique + 1 outgoing timely link.

References

1 M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing omega with weak reliability and synchrony assumptions. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, PODC '03, page 306–314, New York, NY, USA, 2003. Association for Computing Machinery.

2 M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. PODC '04, page 328–337, New York, NY, USA, 2004. Association for Computing Machinery.

3 M. Alfatafta, B. Alkhatib, A. Alquraan, and S. Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 351–368. USENIX Association, Nov. 2020.

4 A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of Network-Partitioning failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 51–68, Carlsbad, CA, Oct. 2018. USENIX Association.

5 M. Bravo, G. Chockler, and A. Gotsman. Making byzantine consensus live. In *DISC*, 2020.

6 M. Bravo, G. Chockler, and A. Gotsman. Liveness and latency of byzantine state-machine replication. *arXiv preprint arXiv:2202.06679*, 2022.

7 C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, apr 1988.

8 M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, apr 1985.

9 C. Jensen, H. Howard, and R. Mortier. Examining raft's behaviour during partial network failures. HAOC '21, page 11–17, New York, NY, USA, 2021. Association for Computing Machinery.

10 O. Naor, M. Baudet, D. Malkhi, and A. Spiegelman. Lumière: Byzantine view synchronization. *CoRR*, abs/1909.05204, 2019.

11 O. Naor and I. Keidar. Expected linear round synchronization: The missing link for linear byzantine SMR. *CoRR*, abs/2002.07539, 2020.

A SMR Synchronizer Correctness

▶ **Lemma 17.** *If a process p_i enters a view v, then there exists a process $p_j \in \mathcal{H}$, a view $v' \geq v$ and a time $t < E_i(v)$ such that p_j sends* `WISH`(v') *at t.*

Proof. Since p_i enters v, $E_i(v)$ is defined and so is $E_{\mathrm{first}}(v) \leq E_i(v)$. Let p_k be the first process to enter v so that $E_k(v) = E_{\mathrm{first}}(v)$. Suppose p_k enters v upon receipt of an `ENTER`(v) message. Then there exists a process p_l and a time $t_l < E_k(v)$ such that p_l sends `ENTER`(v) at t_l. Lines 5 and 20 imply that $p_l.\mathsf{view}(t_l) = v$ and thus $E_{\mathrm{first}}(v) \leq E_l(v) \leq t_l < E_k(v)$, which is a contradiction.

Therefore, $p_k.\mathsf{views}(E_k(v))$ includes $f+1$ entries $\geq v$. Since there exist at most f processes not in $\mathcal{H}$, there exists a process $p_j \in \mathcal{H}$, a view $v' \geq v$ and a time $t < E_k(v) \leq E_i(v)$ such that p_j sends `WISH`(v') at t. ◀

▶ **Lemma 18.** *If a process p_i sends* `WISH`$(v+1)$ *at a time t, then $A_i(v)\downarrow \wedge A_i(v) \leq t$.*

Proof. Let t, $v \geq 0$ and p_i be such that p_i sends `WISH`$(v+1)$ at t. Then at t, p_i executes either line 2 or 7, and thus $p_i.\mathsf{view}(t) = v$.

If p_i executes line 2, then p_i attempts to advance from v at the time t and thus $A_i(v)\downarrow$ and $A_i(v) = t$. Suppose now that p_i executes line 7. Then $E_i(v)\downarrow$ and satisfies $E_i(v) < t \wedge p_i.\mathsf{view}(E_i(v)) = v$. Since $p_i.\mathsf{advanced}(E_i(v)) =$ `FALSE` and $p_i.\mathsf{advanced}(t) =$ `TRUE`, there exists a time t' such that $E_i(v) < t' < t$ at which p_i executes the handler at line 1. Since $p_i.\mathsf{view}$ is non-decreasing and it is equal to v at $E_i(v)$ as well as t, $p_i.\mathsf{view}(t') = v$. Therefore, p_i attempts to advance from v at t' and thus $A_i(v)\downarrow$ and $A_i(v) = t' < t$. ◀

▶ **Lemma 19.** *For all views $v, v' > 0$, if a process sends* `WISH`(v) *before sending* `WISH`(v')*, then $v \leq v'$.*

Proof. Let s and s' be the times at which a process p_i sends `WISH`(v) and `WISH`(v') respectively. Notice that $v = p_i.\mathsf{view}(s) + 1$ and $v' = p_i.\mathsf{view}(s') + 1$. Since $p_i.\mathsf{view}$ is non-decreasing, as guaranteed by lines 9 and 14, $p_i.\mathsf{view}(s) \leq p_i.\mathsf{view}(s')$. Therefore, $v = p_i.\mathsf{view}(s) + 1 \leq p_i.\mathsf{view}(s') + 1 = v'$. ◀

▶ **Lemma 20.** *For all processes $p_i \in \mathcal{H}$, times $t \geq \mathsf{GST} + \rho$, and views v, if p_i sends* `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 either sends* `ENTER`(v') *or* `WISH`(v') *at t.*

Proof. Let $s \leq t$ be the time at which p_i sends `WISH`(v). If $s \geq \mathsf{GST}$, then 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 retransmission code in line 4 at s'.

If $p_i.\mathsf{advanced}(s') =$ `TRUE`, then there exists a view v' such that p_i sends `WISH`(v') at s' by executing the code in line 7. By Lemma 6, $v' \geq v$. Suppose now that $p_i.\mathsf{advanced}(s') =$ `FALSE`. Since $p_i.\mathsf{advanced}(s) =$ `TRUE`, there exists a time s'' satisfying $s < s'' < s'$ such that p_i executes the code in line 18 at s''. Therefore, lines 9 and 14 guarantee that $v = p_i.\mathsf{view}(s) + 1 \leq p_i.\mathsf{view}(s'')$. Since $p_i.\mathsf{view}$ is non-decreasing, $p_i.\mathsf{view}(s'') \leq p_i.\mathsf{view}(s')$, and thus the `ENTER`$(v')$ message that p_i sends at s' satisfies $v' = p_i.\mathsf{view}(s') \geq p_i.\mathsf{view}(s'') \geq v$. ◀

B SMR Protocol Correctness

▶ **Lemma 21.** *Let p_i be a correct process in view v that never enters a view higher than v. If p_i predelivers only finitely many messages or it never predelivers a message that it has broadcast while in v, then it eventually calls* advance *in v.*

Proof. Upon entering v the process p_i starts timer_recovery (line 39). If the timer expires, then p_i calls advance in v (line 67).

Assume the timer does not expire. Then p_i stops the timer at line 38, 51, 60 or 63. The first is impossible, as this together with Monotonicity would imply that p_i enters a higher view. If p_i stops the timer at line 63, then it calls advance in v (line 67). Assume now that p_i stops the timer at line 51 or 60. Then p_i starts the timer timer_commit (lines 52 and 61). If the timer expires, then p_i calls advance in v (line 67). Then p_i stops the timer at line 38, 63 or 28. The first is impossible, as this together with Monotonicity would imply that p_i enters a higher view. If p_i stops the timer at line 63, then p_i calls advance in v (line 67). Then p_i stops the timer at line 28. However, upon doing so, p_i restarts the timer (line 30). Therefore, for the timer timer_commit not to expire, p_i must execute the code in line 28 infinitely many times, and thus, it must predeliver infinitely many messages. Hence, if p_i predelivers only finitely many messages, then its timer timer_commit eventually expires causing p_i to call advance in v (line 67).

Consider now the case in which p_i broadcasts a message m while in v. Upon doing so p_i starts timer_delivery$[m]$ (line 8). If the timer expires, then p_i calls advance in v (line 67). Assume the timer does not expire. Then p_i stops the timer at line 38, 63, or 29. The first is impossible, as this together with Monotonicity would imply that p_i enters a higher view. If p_i stops the timer at line 63, then it calls advance in v (line 67). Assume now that p_i stops the timer at line 29. Then, p_i must predeliver m. Hence, if there exists a message m broadcast by p_i while in v that it never predelivers, its timer timer_delivery$[m]$ eventually expires causing p_i to call advance in v (line 67). ◀

▶ **Lemma 22.** *Consider a view $v \geq \mathcal{V}$ such that $E^{\mathcal{H}}_{\text{first}}(v) \geq \mathsf{GST}$ and $\texttt{leader}(v) = p_c$. If at each process $p_i \in \mathcal{H}$ that enters v we have $\mathsf{dur\_recovery}_i(v) > 5\delta$, $\mathsf{dur\_delivery}_i(v) > 6\delta$ and $\mathsf{dur\_commit}_i(v) > \rho + 6\delta$, then no process in $\mathcal{H}$ calls* advance *in v.*

Proof. Let p_i be the first process in $\mathcal{H}$ to call advance in v. By Lemma 12, we have $\mathsf{dur\_recovery}_i(v) \leq 5\delta$, $\mathsf{dur\_delivery}_i(v) \leq 6\delta$ or $\mathsf{dur\_commit}_i(v) \leq \rho+6\delta$. This contradicts the assumption that $\mathsf{dur\_recovery}_i(v) > 5\delta$, $\mathsf{dur\_delivery}_i(v) > 6\delta$ and $\mathsf{dur\_commit}_i(v) > \rho + 6\delta$. Thus, no process in $\mathcal{H}$ calls advance in v. ◀

C An Impossibility Result for an f-clique + 1 Outgoing Link

Assume a model where there is a time after which there are f correct processes fully connected by timely links and one extra outcoming timely link from the clique to an $(f + 1)$st correct process.

▶ **Theorem 23.** *There does not exist a deterministic algorithm implementing consensus in the above model.*

Proof. Suppose for the sake of contradiction that there exists such an algorithm. Let G_1 be the set $\{1, \dots, f\}$, G_2 be the set $\{f + 1\}$ and G_3 be the set $\{f + 2, \dots, n\}$.

Let C_1 be a configuration where $G_1 \cup G_2$ form the majority stipulated by the system model where processes from G_1 constitute a clique and there is a timely link from the clique to the process $f + 1$. Furthermore, assume that processes from G_3 are initially crashed and that any other link from/to process $f + 1$ drop all messages. Let α_1 be an execution of the system in C_1 that contains an invocation *propose*(v_1) by process 1 and no other invocations. By Termination, a process from G_1 eventually decides with a matching *decide*(v_1). Let α'_1 be the prefix of α_1 ending with *decide*(v_1).

Let C_2 be a configuration where $G_3 \cup G_2$ form the majority stipulated by the system model where processes from G_3 constitute a clique and there is a timely link from the clique to the process $f+1$. Furthermore, assume that processes from G_1 are initially crashed and that any other link from/to from process $f+1$ drop all messages. Let α_2 be an execution of the system in C_2 that contains an invocation $\mathit{propose}(v_2)$ by process n with $v_2 \neq v_1$ and no other invocations. By Termination, a process from G_3 eventually decides with a matching $\mathit{decide}(v_2)$. Let α'_2 be the prefix of α_2 ending with $\mathit{decide}(v_2)$.

Let $\alpha''_1 = \alpha'_1|_{G_1}$ and $\alpha''_2 = \alpha'_2|_{G_2}$. Since there is no communication from the process $f+1$ into G_1, the actions by processes in G_1 are completely independent from the actions by process $f+1$. Therefore, the result α''_1 of removing from α'_1 all the actions by process $f+1$ is a valid execution. Similarily, α''_2 is a valid execution.

Let C_3 be a configuration where G_1 and G_3 form cliques made out of timely links. Also, assume that there is a timely link from G_1 to process $f+1$ and a timely link from G_3 to process $f+1$. Finally, assume that every link between G_1 and G_3 as well as any other link from/to process $f+1$ drop all messages. Note that this is allowed by the system model. Let α be an execution of the system in C_3 that begins with all the activity from α''_1 followed by all the activity from α''_2. Also, assume that all actions in α occur before GST. Since the processes in each group, G_1 or G_2, behave independently of the processes in the other group in α''_1 and α''_2, respectively, this is a valid execution. Furthermore, for any process $p_i \in G_1 \cup G_2$, $\alpha|_{p_i} = \alpha''_1|_{p_i}$ and thus α is indistinguishable from α''_1 to the processes in $G_1 \cup G_2$. Similarily, for any process $p_i \in G_3 \cup G_2$, $\alpha|_{p_i} = \alpha''_2|_{p_i}$ and thus α is indistinguishable from α''_2 to the processes in $G_3 \cup G_2$. Finally, since $\mathit{decide}(v_1)$ occurs in α''_1 and $\mathit{decide}(v_2)$ occurs in α''_2, then both $\mathit{decide}(v_1)$ and $\mathit{decide}(v_2)$ occur in α. This violates the Agreement property, thus yielding a contradiction. ◀