Distributed Computing
https://doi.org/10.1007/s00446-023-00458-w

Byzantine consensus is $\Theta(n^2)$: the Dolev-Reischuk bound is tight even in partial synchrony!

Pierre Civit[1] · Muhammad Ayaz Dzulfikar[2] · Seth Gilbert[2] · Vincent Gramoli[3,4] · Rachid Guerraoui[5] · Jovan Komatovic[5] · Manuel Vidigueira[5]

Received: 29 January 2023 / Accepted: 8 November 2023

Abstract

The Dolev-Reischuk bound says that any deterministic Byzantine consensus protocol has (at least) quadratic (in the number of processes) communication complexity in the worst case: given a system with n processes and at most $f < n/3$ failures, any solution to Byzantine consensus exchanges $\Omega(n^2)$ words, where a word contains a constant number of values and signatures. While it has been shown that the bound is tight in synchronous environments, it is still unknown whether a consensus protocol with quadratic communication complexity can be obtained in partial synchrony where the network alternates between (1) asynchronous periods, with unbounded message delays, and (2) synchronous periods, with δ-bounded message delays. Until now, the most efficient known solutions for Byzantine consensus in partially synchronous settings had cubic communication complexity (e.g., HotStuff, binary DBFT). This paper closes the existing gap by introducing SQuad, a partially synchronous Byzantine consensus protocol with $O(n^2)$ worst-case communication complexity. In addition, SQuad is optimally-resilient (tolerating up to $f < n/3$ failures) and achieves $O(f \cdot \delta)$ worst-case latency complexity. The key technical contribution underlying SQuad lies in the way we solve *view synchronization*, the problem of bringing all correct processes to the same view with a correct leader for sufficiently long. Concretely, we present RareSync, a view synchronization protocol with $O(n^2)$ communication complexity and $O(f \cdot \delta)$ latency complexity, which we utilize in order to obtain SQuad.

Keywords Optimal Byzantine consensus · Communication complexity · Latency complexity

1 Introduction

Byzantine consensus [1] is a fundamental distributed computing problem. In recent years, it has become the target of widespread attention due to the advent of blockchain [2–4] and decentralized cloud computing [5], where it acts as a key primitive. The demand of these contexts for high performance has given a new impetus to research towards Byzantine consensus with optimal communication guarantees.

Intuitively, Byzantine consensus enables processes to agree on a common value despite Byzantine failures. Formally, each process is either correct or faulty; correct

✉ Pierre Civit
pierrecivit@gmail.com

✉ Jovan Komatovic
jovan.komatovic@epfl.ch

✉ Manuel Vidigueira
manuel.ribeirovidigueira@epfl.ch

Muhammad Ayaz Dzulfikar
ayaz.dzulfikar@u.nus.edu

Seth Gilbert
seth.gilbert@comp.nus.edu.sg

Vincent Gramoli
vincent.gramoli@sydney.edu.au

Rachid Guerraoui
rachid.guerraoui@epfl.ch

1 LIP6, Sorbonne University, 4 Place Jussieu, 75005 Paris, France

2 Department of Computer Science, National University of Singapore (NUS), 21 Lower Kent Ridge Rd, Singapore 119077, Singapore

3 CSRG, University of Sydney, 1, Cleveland St., Sydney, NSW 2006, Australia

4 Redbelly Network,, Pty Ltd 304/74 Pitt St., Sydney, NSW 2000, Australia

5 DCL, Ecole Polytechnique Fédérale de Lausanne (EPFL), Rte Cantonale, 1015 Lausanne, Switzerland

processes follow a prescribed protocol, whereas faulty processes (up to $0 < f < n/3$) can arbitrarily deviate from it. Each correct process *proposes* a value, and should eventually *decide* a value (no more than once). The following properties are guaranteed:

- *Validity:* If all correct processes propose the same value, then only that value can be decided by a correct process.
- *Agreement:* No two correct processes decide different values.
- *Termination:* All correct processes eventually decide.

The celebrated Dolev-Reischuk bound [6] on message complexity implies that any deterministic solution of the Byzantine consensus problem incurs $\Omega(n^2)$ communication complexity: correct processes must send $\Omega(n^2)$ words in the worst case, where a word contains a constant number of values and signatures. It has been shown that the bound is tight in synchronous environments [7, 8]. However, for the partially synchronous environments [9] in which the network initially behaves asynchronously and starts being synchronous with δ-bounded message delays only after some unknown Global Stabilization Time (GST), no Byzantine consensus protocol achieving quadratic communication complexity is known.[1] (Importantly, the communication complexity of partially synchronous Byzantine consensus protocols only counts words sent after GST as the number of words sent before GST is unbounded [11]). Therefore, the question remains whether a partially synchronous Byzantine consensus with quadratic communication complexity exists [12]. Until now, the most efficient known solutions in partially synchronous environments had $O(n^3)$ communication complexity (e.g., HotStuff [13], binary DBFT [2]).

We close the gap by introducing SQuad, a partially synchronous Byzantine consensus protocol with $O(n^2)$ worst-case communication complexity, matching the Dolev-Reischuk [6] bound. In addition, SQuad is optimally-resilient (tolerating up to $f < n/3$ failures) and achieves $O(f \cdot \delta)$ worst-case latency (measured from GST onwards).

1.1 Partially synchronous "leader-based" Byzantine consensus

Partially synchronous "leader-based" consensus protocols [13–16] operate in *views*, each with a designated leader whose responsibility is to drive the system towards a decision. If a process does not decide in a view, the process moves to the next view with a different leader and tries again. Once all correct processes overlap in the same view with a correct leader for sufficiently long, a decision is reached. Sadly, ensuring such an overlap is non-trivial; for example, processes can start executing the protocol at different times or their local clocks may drift before GST, thus placing them in views which are arbitrarily far apart.

Typically, these protocols contain two independent modules:

1. View core: The core of the protocol, responsible for executing the protocol logic of each view.
2. View synchronizer: Auxiliary to the view core, responsible for "moving" processes to new views with the goal of ensuring a sufficiently long overlap to allow the view core to decide.

Immediately after GST, the view synchronizer brings all correct processes together to the view of the most advanced correct process and keeps them in that view for sufficiently long. At this point, if the leader of the view is correct, the processes decide. Otherwise, they "synchronously" transit to the next view with a different leader and try again. Figure 1 illustrates this mechanism in HotStuff [13]. In summary, the communication complexity of such consensus protocols can be approximated by $n \cdot C + S$, where:

- C denotes the maximum number of words a correct process sends while executing its view core during $[GST, t_d]$, where t_d is the first time by which all correct processes have decided,[2] and
- S denotes the communication complexity of the view synchronizer during $[GST, t_d]$.

Since the adversary can corrupt up to f processes, correct processes must transit through at least $f + 1$ views after GST, in the worst case, before reaching a correct leader. In fact, PBFT [15] and HotStuff [13] show that passing through $f + 1$ views is sufficient to reach a correct leader. Furthermore, HotStuff employs the "leader-to-all, all-to-leader" communication pattern in each view. As (1) each process is the leader of at most one view during $[GST, t_d]$, and (2) a process sends $O(n)$ words in a view if it is the leader of the view, and $O(1)$ words otherwise, HotStuff achieves $C = 1 \cdot O(n) + f \cdot O(1) = O(n)$. Unfortunately, $S = (f + 1) \cdot O(n^2) = O(n^3)$ in HotStuff due to "all-to-all" communication exploited by its view synchronizer in *every* view.[3] Thus, $S = O(n^3)$ dominates the communication complexity of HotStuff, preventing it from matching the Dolev-Reischuk bound. If we could design a consensus algorithm for which $S = O(n^2)$ while preserving $C = O(n)$, we

[1] No deterministic protocol solves Byzantine consensus in a completely asynchronous environment [10].

[2] "d" in "t_d" stands for *d*ecide/*d*ecision.

[3] While HotStuff [13] does not explicitly state how the view synchronization is achieved, we have that $S = O(n^3)$ in Diem BFT [14], which is a mature implementation of the HotStuff protocol.

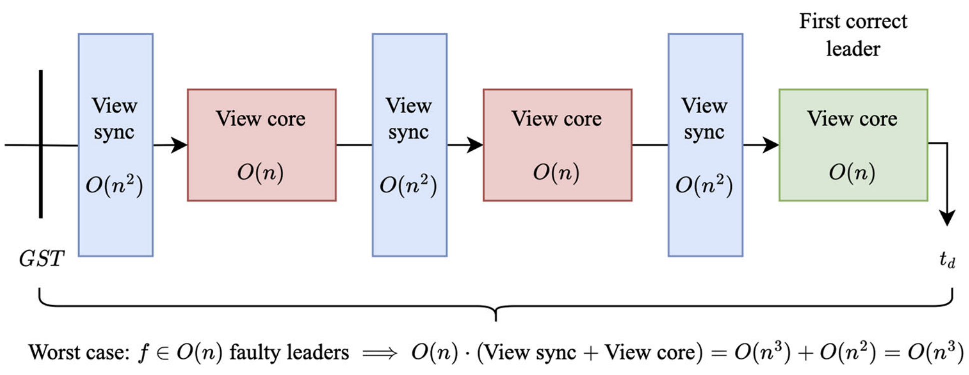
View sync
$O(n^2)$
View core
$O(n)$
View sync
$O(n^2)$
View core
$O(n)$
View sync
$O(n^2)$
First correct leader
View core
$O(n)$
GST
t_d
Worst case: $f \in O(n)$ faulty leaders $\implies O(n) \cdot (\text{View sync} + \text{View core}) = O(n^3) + O(n^2) = O(n^3)$

Fig. 1 Overview of HotStuff [13]: Processes change views via the view synchronizer. After GST, once a correct leader is reached all correct processes are guaranteed to decide (by time t_d). In the worst case, the first $f \in O(n)$ leaders after GST are faulty and $O(n)$ view synchronizations are executed, resulting in $O(n^3)$ communication complexity

would obtain a Byzantine consensus protocol with optimal communication complexity. The question is if a view synchronizer achieving $S = O(n^2)$ in partial synchrony exists.

1.2 Warm-up: View synchronization in complete synchrony

Solving the synchronization problem in a completely synchronous environment is not hard. As all processes start executing the protocol at the same time and their local clocks do not drift, the desired overlap can be achieved without any communication: processes stay in each view for the fixed, overlap-required time. However, this simple method *cannot* be used in a partially synchronous setting as it is neither guaranteed that all processes start at the same time nor that their local clocks do not drift (before GST). Still, the observation that, if the system is completely synchronous, processes are not required to communicate in order to synchronize plays a crucial role in developing our view synchronizer which achieves $O(n^2)$ communication complexity in partially synchronous environments.

1.3 RareSync

The main technical contribution of this work is RareSync, a partially synchronous view synchronizer that achieves synchronization within $O(f \cdot \delta)$ time after GST, and has $O(n^2)$ worst-case communication complexity. In a nutshell, RareSync adapts the "no-communication" technique of synchronous view synchronizers to partially synchronous environments.

Namely, RareSync groups views into *epochs*; each epoch contains $f + 1$ sequential views. Instead of performing "all-to-all" communication in each view (like the "traditional" view synchronizers [14]), RareSync performs a *single* "all-to-all" communication step per epoch. Specifically, *only* at the end of each epoch do all correct processes communicate to enable further progress. Once a process has entered an epoch, the process relies *solely* on its local clock (without any communication) to move forward to the next view within the epoch.

Let us give a (rough) explanation of how RareSync ensures synchronization. Let E be the smallest epoch entered by *all* correct processes at or after GST; let the first correct process enter E at time $t_E \geq GST$. Due to (1) the "all-to-all" communication step performed at the end of the previous epoch $E-1$, and (2) the fact that message delays are bounded by a known constant δ after GST, all correct processes enter E by time $t_E + \delta$. Hence, from the epoch E onward, processes do not need to communicate in order to synchronize: it is sufficient for processes to stay in each view for $\delta + \Delta$ time to achieve Δ-time overlap. In brief, RareSync uses communication to synchronize processes, while relying on local timeouts (and not communication!) to keep them synchronized.

1.4 SQuad

The second contribution of our work is SQuad, an optimally-resilient partially synchronous Byzantine consensus protocol with (1) $O(n^2)$ worst-case communication complexity, and (2) $O(f \cdot \delta)$ worst-case latency complexity. The view core module of SQuad is the same as that of HotStuff; as its view synchronizer, SQuad uses RareSync. The combination of HotStuff's view core and RareSync ensures that $C = O(n)$ and $S = O(n^2)$. By the aforementioned complexity formula, SQuad achieves $n \cdot O(n) + O(n^2) = O(n^2)$ communication complexity. SQuad's linear latency is a direct consequence of RareSync's ability to synchronize processes within $O(f \cdot \delta)$ time after GST.

1.5 Roadmap

We discuss related work in Sect. 2. In Sect. 3, we define the system model. We introduce RareSync in Sect. 4. In Sect. 5, we present SQuad. We conclude the paper in Sect. 6. Detailed proofs of the most basic properties of RareSync are delegated to Appendix B.

2 Related work

In this section, we discuss existing results in two related contexts: synchronous networks and randomized algorithms. In addition, we discuss some precursor (and concurrent) results to our own.

2.1 Synchronous networks

The first natural question is whether we can achieve synchronous Byzantine agreement with optimal latency and optimal communication complexity. Momose and Ren answer that question in the affirmative, giving a synchronous Byzantine agreement protocol with optimal $n/2$ resiliency, optimal $O(n^2)$ worst-case communication complexity and optimal $O(f)$ worst-case latency [8]. Optimality follows from two lower bounds: Dolev and Reischuk show that any Byzantine consensus protocol has an execution with quadratic communication complexity [6]; Dolev and Strong show that any synchronous Byzantine consensus protocol has an execution with $f+1$ rounds [17]. Various other works have tackled the problem of minimizing the latency of Byzantine consensus [18–20].

2.2 Randomization

A classical approach to circumvent the FLP impossibility [10] is using randomization [21], where termination is not ensured deterministically. Exciting recent results by Abraham et al. [22] and Lu et al. [23] give fully asynchronous randomized Byzantine consensus with optimal $n/3$ resiliency, optimal $O(n^2)$ expected communication complexity and optimal $O(1)$ expected latency complexity. Spiegelman [11] took a neat *hybrid* approach that achieved optimal results for both synchrony and randomized asynchrony simultaneously: if the network is synchronous, his algorithm yields optimal (deterministic) synchronous complexity; if the network is asynchronous, it falls back on a randomized algorithm and achieves optimal expected complexity.

Recently, it has been shown that even randomized Byzantine agreement requires $\Omega(n^2)$ expected communication complexity, at least for achieving guaranteed safety against an *adaptive adversary* in an asynchronous setting or against a *strongly rushing adaptive adversary* in a synchronous setting [22, 24]. (See the papers for details). Amazingly, it is possible to break the $O(n^2)$ barrier by accepting a non-zero (but $o(1)$) probability of disagreement [25–27].

2.3 Authentication

Most of the results above are *authenticated*: they assume a trusted setup phase[4] wherein devices establish and exchange cryptographic keys; this allows for messages to be signed in a way that proves who sent them. Recently, many of the communication-efficient agreement protocols (such as [22, 23]) rely on *threshold signatures* (such as [28]). The Dolev-Reischuk [6] lower bound shows that quadratic communication is needed even in such a case (as it looks at the message complexity of authenticated agreement).

Among deterministic, non-authenticated Byzantine agreement protocols, DBFT [2] achieves $O(n^3)$ communication complexity. For randomized non-authenticated Byzantine agreement protocols, Abraham et al. [29] generalize and refine the findings of Mostefaoui et al. [30] to achieve $O(n^2)$ communication complexity; however they assume a weak common coin, for which an implementation with $O(n^2)$ communication complexity may also require signatures (as far as we are aware, known implementations without signatures, such as [31, 32], have higher than $O(n^2)$ communication complexity).

We note that it is possible to (1) work towards an authenticated setting from a non-authenticated one by rolling out a public key infrastructure (PKI) [33–35], (2) setting up a threshold scheme [36] without a *trusted dealer*, and (3) asynchronously emulating a perfect common coin [37] used by randomized Byzantine consensus protocols [22, 23, 30, 38], or implementing it without signatures [31, 32].

2.4 Other related work

In this paper, we focus on the partially synchronous setting [9], where the question of optimal communication complexity of Byzantine agreement has remained open. The question can be addressed precisely with the help of rigorous frameworks [39–41] that were developed to express partially synchronous protocols using a round-based paradigm. More specifically, state-of-the-art partially synchronous BFT protocols [13, 14, 16, 42] have been developed within a view-based paradigm with a rotating leader, e.g., the seminal PBFT protocol [15]. While many approaches improve the complexity for some optimistic scenarios [43–47], none of them were able to reach the quadratic worst-case Dolev-Reischuk bound.

The problem of view synchronization was defined in [48]. An existing implementation of this abstraction [42] was based on Bracha's double-echo reliable broadcast at each view, inducing a cubic communication complexity in total. This communication complexity has been reduced for some

[4] A trusted setup phase is notably different from randomized algorithms where randomization is used throughout.

optimistic scenarios [48] and in terms of *expected* complexity [49]. The problem has been formalized more precisely in [50] to facilitate formal verification of PBFT-like protocols.

It might be worthwhile highlighting some connections between the view synchronization abstraction and the leader election abstraction Ω [51, 52], capturing the weakest failure detection information needed to solve consensus (and extended to the Byzantine context in [53]). Leaderless partially synchronous Byzantine consensus protocols have also been proposed [54], somehow indicating that the notion of a leader is not necessary in the mechanisms of a consensus protocol, even if Ω is the weakest failure detector needed to solve the problem. Clock synchronization [55, 56] and view synchronization are orthogonal problems.

2.5 Concurrent research

We have recently discovered concurrent and independent research by Lewis-Pye [57]. Lewis-Pye appears to have discovered a similar approach to the one that we present in this paper, giving an algorithm for state machine replication in a partially synchronous model with quadratic message complexity. As in this paper, Lewis-Pye makes the key observation that we do not need to synchronize in every view; views can be grouped together, with synchronization occurring only once every fixed number of views. This yields essentially the same algorithmic approach. Lewis-Pye focuses on state machine replication, instead of Byzantine agreement (though state machine replication is implemented via repeated Byzantine agreement). The other useful property of his algorithm is *optimistic responsiveness*, which applies to the multi-shot case and ensures that, in good portions of the executions, decisions happen as quickly as possible. We encourage the reader to look at [57] for a different presentation of a similar approach.

Moreover, the similar approach to ours and Lewis-Pye's has been proposed in the first version of HotStuff [58]: processes synchronize once per *level*, where each level consists of n views. The authors mention that this approach guarantees the quadratic communication complexity; however, this claim was not formally proven in their work. The claim was dropped in later versions of HotStuff (including the published version). We hope readers of our paper will find an increased appreciation of the ideas introduced by HotStuff.

3 System model

3.1 Processes

We consider a static set $\{P_1, P_2, \dots, P_n\}$ of $n = 3f + 1$ processes out of which at most f can be Byzantine, i.e., can behave arbitrarily. If a process is Byzantine, the process is *faulty*; otherwise, the process is *correct*. Processes communicate by exchanging messages over an authenticated point-to-point network. The communication network is *reliable:* if a correct process sends a message to a correct process, the message is eventually received. We assume that processes have local hardware clocks. Furthermore, we assume that local steps of processes take zero time, as the time needed for local computation is negligible compared to message delays. Finally, we assume that no process can take infinitely many steps in finite time.

3.2 Partial synchrony

We consider the partially synchronous model introduced in [9]. For every execution, there exists a Global Stabilization Time (GST) and a positive duration δ such that message delays are bounded by δ after GST. Furthermore, GST is not known to processes, whereas δ is known to processes. We assume that all correct processes start executing their protocol by GST. The hardware clocks of processes may drift arbitrarily before GST, but do not drift thereafter.

3.3 Cryptographic primitives

We assume a (k, n)-threshold signature scheme [28], where $k = 2f + 1 = n - f$. In this scheme, each process holds a distinct private key and there is a single public key. Each process P_i can use its private key to produce an unforgeable partial signature of a message m by invoking $\mathit{ShareSign}_i(m)$. A partial signature $\mathit{tsignature}$ of a message m produced by a process P_i can be verified by $\mathit{ShareVerify}_i(m, \mathit{tsignature})$. Finally, set $S = \{\mathit{tsignature}_i\}$ of partial signatures, where $|S| = k$ and, for each $\mathit{tsignature}_i \in S$, $\mathit{tsignature}_i = \mathit{ShareSign}_i(m)$, can be combined into a *single* (threshold) signature by invoking $\mathit{Combine}(S)$; a combined signature $\mathit{tcombined}$ of message m can be verified by $\mathit{CombinedVerify}(m, \mathit{tcombined})$. Where appropriate, invocations of $\mathit{ShareVerify}(\cdot)$ and $\mathit{CombinedVerify}(\cdot)$ are implicit in our descriptions of protocols. P_Signature and T_Signature denote a partial signature and a (combined) threshold signature, respectively. A formal treatment of the aforementioned threshold signature scheme is relegated to Appendix A.

3.4 Complexity of Byzantine consensus

Let Consensus be a partially synchronous Byzantine consensus protocol and let $\mathcal{E}(\text{Consensus})$ denote the set of all possible executions. Let $\alpha \in \mathcal{E}(\text{Consensus})$ be an execution and $t_d(\alpha)$ be the first time by which all correct processes have decided in α.

A *word* contains a constant number of signatures and values. Each message contains at least a single word. We define the communication complexity of α as the number of words sent in messages by all correct processes during the time period $[GST, t_d(\alpha)]$; if $GST > t_d(\alpha)$, the communication complexity of α is 0. The latency complexity of α is $\max(0, t_d(\alpha) - GST)$.

The *communication complexity* of Consensus is defined as

$$\max_{\alpha \in \mathcal{E}(\text{Consensus})} \Big\{\text{communication complexity of } \alpha\Big\}.$$

Similarly, the *latency complexity* of Consensus is defined as

$$\max_{\alpha \in \mathcal{E}(\text{Consensus})} \Big\{\text{latency complexity of } \alpha\Big\}.$$

We underline that the number of words sent by correct processes before GST is unbounded in any partially synchronous Byzantine consensus protocol [11]. Moreover, not a single correct process is guaranteed to decide before GST in any partially synchronous Byzantine consensus protocol [10]; that is why the latency complexity of such protocols is measured from GST.

Note on communication complexity. The communication complexity metric as defined above is sometimes referred to as *word complexity* [59, 60]. Another traditional communication complexity metric is *bit complexity* [7, 61], which counts bits (and not words) sent by correct processes. Note that the Dolev-Reischuk lower bound on exchanged messages implies $\Omega(n^2)$ (with $f \in \Omega(n)$) lower bound on both word (a message contains at least one word) and bit complexity (a message contains at least one bit). We underline that QUAD is not optimal with respect to the bit complexity as it achieves $O(n^2L + n^2\kappa)$ bit complexity, where L is the size of the values and κ is the security parameter (e.g., size of a signature).

4 RARESYNC

This section presents RARESYNC, a partially synchronous view synchronizer that achieves synchronization within $O(f \cdot \delta)$ time after GST, and has $O(n^2)$ worst-case communication complexity. First, we define the problem of view synchronization (Sect. 4.1). Then, we describe RARESYNC, and present its pseudocode (Sect. 4.2). Finally, we reason about RARESYNC's correctness and complexity (Sect. 4.3) before presenting a formal proof (Sect. 4.4).

4.1 Problem definition

View synchronization is defined as the problem of bringing all correct processes to the same view with a correct leader for sufficiently long [48–50]. More precisely, let $\text{View} = \{1, 2, \ldots\}$ denote the set of views. For each view $v \in \text{View}$, we define $\text{leader}(v)$ to be a process that is the *leader* of view v. The view synchronization problem is associated with a predefined time $\Delta > 0$, which denotes the desired duration during which processes must be in the same view with a correct leader in order to synchronize. View synchronization provides the following interface:

- **Indication** advance(View v): The process advances to a view v.

We say that a correct process *enters* a view v at time t if and only if the $\text{advance}(v)$ indication occurs at time t. Moreover, a correct process *is in view* v between the time t (including t) at which the $\text{advance}(v)$ indication occurs and the time t' (excluding t') at which the next $\text{advance}(v' \neq v)$ indication occurs. If an $\text{advance}(v' \neq v)$ indication never occurs, the process remains in the view v from time t onward.

Next, we define a *synchronization time* as a time at which all correct processes are in the same view with a correct leader for (at least) Δ time.

Definition 1 (Synchronization time) Time t_s is a *synchronization time* if (1) all correct processes are in the same view v from time t_s to (at least) time $t_s + \Delta$, and (2) $\text{leader}(v)$ is correct.

View synchronization ensures the *eventual synchronization* property which states that there exists a synchronization time at or after GST.

4.1.1 Complexity of view synchronization

Let Synchronizer be a partially synchronous view synchronizer and let $\mathcal{E}(\text{Synchronizer})$ denote the set of all possible executions. Let $\alpha \in \mathcal{E}(\text{Synchronizer})$ be an execution and $t_s(\alpha)$ be the first synchronization time at or after GST in α ($t_s(\alpha) \geq GST$). We define the communication complexity of α as the number of words sent in messages by all correct processes during the time period $[GST, t_s(\alpha) + \Delta]$. The latency complexity of α is $t_s(\alpha) + \Delta - GST$.

The *communication complexity* of Synchronizer is defined as

$$\max_{\alpha \in \mathcal{E}(\text{Synchronizer})} \Big\{\text{communication complexity of } \alpha\Big\}.$$

Similarly, the *latency complexity* of Synchronizer is defined as

$$\max_{\alpha \in \mathcal{E}(\text{Synchronizer})} \Big\{\text{latency complexity of } \alpha\Big\}.$$

Algorithm 1 RARESYNC: Variables (for process P_i), constants, and functions

```
1: Variables:
2:     Epoch $epoch_i \leftarrow 1$                                   ▷ current epoch
3:     View $view_i \leftarrow 1$   ▷ current view within the current epoch; $view_i \in [1, f+1]$
4:     Timer $view\_timer_i$                  ▷ measures the duration of the current view
5:     Timer $dissemination\_timer_i$           ▷ measures the duration between two communication steps
6:     T_Signature $epoch\_sig_i \leftarrow \perp$           ▷ proof that $epoch_i$ can be entered
7: Constants:
8:     Time $view\_duration = \Delta + 2\delta$                    ▷ duration of each view
9: Functions:
10:     leader(View $v$) $\equiv P_{(v \bmod n)+1}$                  ▷ a round-robin function
```

Algorithm 2 RARESYNC: Pseudocode (for process P_i)

```
1: upon init:                                              ▷ start of the protocol
2:     $view\_timer_i$.measure($view\_duration$)  ▷ measure the duration of the first view
3:     trigger advance(1)                                  ▷ enter the first view
4: upon $view\_timer_i$ expires:
5:     if $view_i < f + 1$:     ▷ check if the current view is not the last view of the current epoch
6:         $view_i \leftarrow view_i + 1$
7:         View $view\_to\_advance \leftarrow (epoch_i - 1) \cdot (f + 1) + view_i$
8:         $view\_timer_i$.measure($view\_duration$)  ▷ measure the duration of the view
9:         trigger advance($view\_to\_advance$)                ▷ enter the next view
10:     else:
11:         ▷ inform other processes that the epoch is completed
12:         broadcast ⟨EPOCH- COMPLETED, $epoch_i$, $ShareSign_i(epoch_i)$⟩
13: upon exists Epoch $e$ such that $e \geq epoch_i$ and ⟨EPOCH- COMPLETED, $e$, P_Signature $sig$⟩ is received from $2f + 1$ processes:
14:     $epoch\_sig_i \leftarrow Combine(\{sig \mid sig$ is received in an EPOCH- COMPLETED message$\})$
15:     $epoch_i \leftarrow e + 1$
16:     $view\_timer_i$.cancel()
17:     $dissemination\_timer_i$.cancel()
18:     $dissemination\_timer_i$.measure($\delta$)      ▷ wait $\delta$ time before broadcasting ENTER- EPOCH
19: upon reception of ⟨ENTER- EPOCH, Epoch $e$, T_Signature $sig$⟩ such that $e > epoch_i$:
20:     $epoch\_sig_i \leftarrow sig$               ▷ $sig$ is a threshold signature of epoch $e - 1$
21:     $epoch_i \leftarrow e$
22: $view\_timer_i$.cancel()
23:     $dissemination\_timer_i$.cancel()
24:     $dissemination\_timer_i$.measure($\delta$)      ▷ wait $\delta$ time before broadcasting ENTER- EPOCH
25: upon $dissemination\_timer_i$ expires:
26:     broadcast ⟨ENTER- EPOCH, $epoch_i$, $epoch\_sig_i$⟩
27:     $view_i \leftarrow 1$                                 ▷ reset the current view to 1
28:     View $view\_to\_advance \leftarrow (epoch_i - 1) \cdot (f + 1) + view_i$
29:     $view\_timer_i$.measure($view\_duration$)      ▷ measure the duration of the view
30:     trigger advance($view\_to\_advance$)     ▷ enter the first view of the new epoch
```

4.2 Protocol

This subsection details RARESYNC (Algorithm 2). In essence, RARESYNC achieves $O(n^2)$ communication complexity and $O(f \cdot \delta)$ latency complexity by exploiting "all-to-all" communication only once per $f + 1$ views instead of once per view.

4.2.1 Intuition

We group views into *epochs*, where each epoch contains $f + 1$ sequential views; $\mathsf{Epoch} = \{1, 2, \ldots\}$ denotes the set of epochs. Processes move through an epoch solely by means of local timeouts (without any communication). However, at the end of each epoch, processes engage in an "all-to-all" communication step to obtain permission to move onto the next epoch: (1) Once a correct process has completed an epoch, it broadcasts a message informing other processes of its completion; (2) Upon receiving $2f + 1$ of such messages, a correct process enters the future epoch. Note that (2) applies to *all* processes, including those in arbitrarily "old" epochs. Overall, this "all-to-all" communication step is the *only* communication processes perform within a single epoch, implying that per-process communication complexity in each epoch is $O(n)$. Figure 2 illustrates the main idea behind RARESYNC.

Roughly speaking, after GST, all correct processes simultaneously enter the same epoch within $O(f \cdot \delta)$ time. After entering the same epoch, processes are guaranteed to synchronize in that epoch, which takes (at most) an additional $O(f \cdot \delta)$ time. Thus, the latency complexity of RARESYNC is $O(f \cdot \delta)$. The communication complexity of RARESYNC is $O(n^2)$ as every correct process executes at most a constant number of epochs, each with $O(n)$ per-process communication, after GST.

4.2.2 Protocol description

We now explain how RARESYNC works. The pseudocode of RARESYNC is given in Algorithm 2, whereas all variables, constants, and functions are presented in Algorithm 1.

We explain RARESYNC's pseudocode (Algorithm 2) from the perspective of a correct process P_i. Process P_i utilizes two timers: $view\_timer_i$ and $dissemination\_timer_i$. A timer has two methods:

1. measure(Time x): After exactly x time as measured by the local clock, an expiration event is received by the host. Note that, as local clocks can drift before GST, x time as measured by the local clock may not amount to x real time (before GST).
2. cancel(): This method cancels all previously invoked measure($\cdot$) methods on that timer, i.e., all pending expiration events (pertaining to that timer) are removed from the event queue.

In RARESYNC, leader($\cdot$) is a round-robin function (line 10 of Algorithm 1).

Once P_i starts executing RARESYNC (line 1), it instructs $view\_timer_i$ to measure the duration of the first view (line 2) and it enters the first view (line 3).

Once $view\_timer_i$ expires (line 4), P_i checks whether the current view is the last view of the current epoch, $epoch_i$ (line 5). If that is not the case, the process advances to the next view of $epoch_i$ (line 9). Otherwise, the process broadcasts an EPOCH- COMPLETED message (line 12) signaling that it has

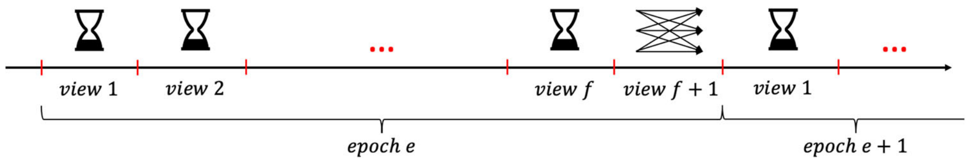

Fig. 2 Intuition behind RARESYNC: Processes communicate only in the last view of an epoch; before the last view, they rely solely on local timeouts

completed $epoch_i$. At this point in time, the process does not enter any view.

If, at any point in time, P_i receives either (1) $2f+1$ EPOCH-COMPLETED messages for some epoch $e \geq epoch_i$ (line 13), or (2) an ENTER-EPOCH message for some epoch $e' > epoch_i$ (line 19), the process obtains a proof that a new epoch $E > epoch_i$ can be entered. However, before entering E and propagating the information that E can be entered, P_i waits δ time (either line 18 or line 24). This δ-waiting step is introduced to limit the number of epochs P_i can enter within any δ time period after GST and is crucial for keeping the communication complexity of RARESYNC quadratic. For example, suppose that processes are allowed to enter epochs and propagate ENTER-EPOCH messages without waiting. Due to an accumulation (from before GST) of ENTER-EPOCH messages for different epochs, a process might end up disseminating an arbitrary number of these messages by receiving them all at (roughly) the same time. To curb this behavior, given that message delays are bounded by δ after GST, we force a process to wait δ time, during which it receives all accumulated messages, before entering the largest known epoch.

Finally, after δ time has elapsed (line 25), P_i disseminates the information that the epoch E can be entered (line 26) and it enters the first view of E (line 30).

4.3 Proof overview

This subsection presents an overview of the proof of the correctness, latency complexity, and communication complexity of RARESYNC.

In order to prove the correctness of RARESYNC, we must show that the eventual synchronization property is ensured, i.e., there is a synchronization time $t_s \geq GST$. For the latency complexity, it suffices to bound $t_s + \Delta - GST$ by $O(f \cdot \delta)$. This is done by proving that synchronization happens within (at most) 2 epochs after GST. As for the communication complexity, we prove that any correct process enters a constant number of epochs during the time period $[GST, t_s + \Delta]$. Since every correct process sends $O(n)$ words per epoch, the communication complexity of RARESYNC is $O(1) \cdot O(n) \cdot n = O(n^2)$. We work towards these conclusions by introducing some key concepts and presenting a series of intermediate results.

A correct process *enters* an epoch e at time t if and only if the process enters the first view of e at time t (either line 3 or line 30). We denote by t_e the first time a correct process enters epoch e.

Result 1: *If a correct process enters an epoch $e > 1$, then (at least) $f+1$ correct processes have previously entered epoch $e-1$.*

The goal of the communication step at the end of each epoch is to prevent correct processes from arbitrarily entering future epochs. In order for a new epoch $e > 1$ to be entered, at least $f+1$ correct processes must have entered and "gone through" each view of the previous epoch, $e-1$. This is indeed the case: in order for a correct process to enter e, the process must either (1) collect $2f+1$ EPOCH-COMPLETED messages for $e-1$ (line 13), or (2) receive an ENTER-EPOCH message for e, which contains a threshold signature of $e-1$ (line 19). In either case, at least $f+1$ correct processes must have broadcast EPOCH-COMPLETED messages for epoch $e-1$ (line 12), which requires them to go through epoch $e-1$. Furthermore, $t_{e-1} \leq t_e$; recall that local clocks can drift before GST.

Result 2: *Every epoch is eventually entered by a correct process.*

By contradiction, consider the greatest epoch ever entered by a correct process, e^*. In brief, every correct process will eventually (1) receive the ENTER-EPOCH message for e^* (line 19), (2) enter e^* after its *dissemination_timer* expires (lines 25 and 30), (3) send an EPOCH-COMPLETED message for e^* (line 12), (4) collect $2f+1$ EPOCH-COMPLETED messages for e^* (line 13), and, finally, (5) enter e^*+1 (lines 15, 18, 25 and 30), resulting in a contradiction. Note that, if $e^* = 1$, no ENTER-EPOCH message is sent: all correct processes enter $e^* = 1$ once they start executing RARESYNC (line 3).

We now define two epochs: e_{max} and $e_{final} = e_{max} + 1$. These two epochs are the main protagonists in the proof of correctness and complexity of RARESYNC.

Definition of e_{max}: *Epoch e_{max} is the greatest epoch entered by a correct process before GST; if no such epoch exists, $e_{max} = 0$.*[5]

Definition of e_{final}: *Epoch e_{final} is the smallest epoch first entered by a correct process at or after GST. Note that $GST \leq t_{e_{final}}$. Moreover, $e_{final} = e_{max} + 1$ (by Result 1).*

Result 3: *For any epoch $e \geq e_{final}$, no correct process broadcasts an* EPOCH-COMPLETED *message for e (line 12) before time $t_e + epoch\_duration$, where $epoch\_duration = (f+1) \cdot view\_duration$.*

[5] *Epoch 0 is considered as a special epoch. Note that $0 \notin$* Epoch, *where* Epoch *denotes the set of epochs (see* Sect. 4.2).

This statement is a direct consequence of the fact that, after GST, it takes exactly $epoch\_duration$ time for a process to go through $f+1$ views of an epoch; local clocks do not drift after GST. Specifically, the earliest a correct process can broadcast an EPOCH- COMPLETED message for e (line 12) is at time $t_e + epoch\_duration$, where t_e denotes the first time a correct process enters epoch e.

Result 4: *Every correct process enters epoch e_{final} by time $t_{e_{final}} + 2\delta$.*

Recall that the first correct process enters e_{final} at time $t_{e_{final}}$. If $e_{final} = 1$, all correct processes enter e_{final} at $t_{e_{final}}$. Otherwise, by time $t_{e_{final}} + \delta$, all correct processes will have received an ENTER- EPOCH message for e_{final} and started the $dissemination\_timer_i$ with $epoch_i = e_{final}$ (either lines 15, 18 or 21, 24). By results 1 and 3, no correct process sends an EPOCH- COMPLETED message for an epoch $\geq e_{final}$ (line 12) before time $t_{e_{final}} + epoch\_duration$, which implies that the $dissemination\_timer$ will not be cancelled. Hence, the $dissemination\_timer$ will expire by time $t_{e_{final}} + 2\delta$, causing all correct processes to enter e_{final} by time $t_{e_{final}} + 2\delta$.

Result 5: *In every view of e_{final}, processes overlap for (at least) Δ time. In other words, there exists a synchronization time $t_s \leq t_{e_{final}} + epoch\_duration - \Delta$.*

By Result 3, no future epoch can be entered before time $t_{e_{final}} + epoch\_duration$. This is precisely enough time for the first correct process (the one to enter e_{final} at $t_{e_{final}}$) to go through all $f+1$ views of e_{final}, spending $view\_duration$ time in each view. Since clocks do not drift after GST and processes spend the same amount of time in each view, the maximum delay of 2δ between processes (Result 4) applies to every view in e_{final}. Thus, all correct processes overlap with each other for (at least) $view\_duration - 2\delta = \Delta$ time in every view of e_{final}. As the $\mathsf{leader}(\cdot)$ function is round-robin, at least one of the $f+1$ views must have a correct leader. Therefore, synchronization must happen within epoch e_{final}, i.e., there is a synchronization time t_s such that $t_{e_{final}} + \Delta \leq t_s + \Delta \leq t_{e_{final}} + epoch\_duration$.

Result 6: $t_{e_{final}} \leq GST + epoch\_duration + 4\delta$.

If $e_{final} = 1$, all correct processes started executing RARESYNC at time GST. Hence, $t_{e_{final}} = GST$. Therefore, the result trivially holds in this case.

Let $e_{final} > 1$; recall that $e_{final} = e_{max} + 1$. (1) By time $GST + \delta$, every correct process receives an ENTER- EPOCH message for e_{max} (line 19) as the first correct process to enter e_{max} has broadcast this message before GST (line 26). Hence, (2) by time $GST + 2\delta$, every correct process enters e_{max}.[6] Then, (3) every correct process broadcasts an EPOCH- COMPLETED message for e_{max} at time $GST + epoch\_duration + 2\delta$ (line 12), at latest. (4) By time $GST + epoch\_duration + 3\delta$, every correct process receives $2f+1$ EPOCH- COMPLETED messages for e_{max} (line 13), and triggers the $\mathsf{measure}(\delta)$ method of $dissemination\_timer$ (line 18). Therefore, (5) by time $GST + epoch\_duration + 4\delta$, every correct process enters $e_{max} + 1 = e_{final}$. Figure 3 depicts this scenario.

Note that for the previous sequence of events *not* to unfold would imply an even lower bound on $t_{e_{final}}$: a correct process would have to receive $2f+1$ EPOCH- COMPLETED messages for e_{max} or an ENTER- EPOCH message for $e_{max} + 1 = e_{final}$ before step (4) (i.e., before time $GST + epoch\_duration + 3\delta$), thus showing that $t_{e_{final}} < GST + epoch\_duration + 4\delta$.

Latency: *Latency complexity of* RARESYNC *is $O(f \cdot \delta)$.*

By Result 5, $t_s \leq t_{e_{final}} + epoch\_duration - \Delta$. By Result 6, $t_{e_{final}} \leq GST + epoch\_duration + 4\delta$. Therefore, $t_s \leq GST + epoch\_duration + 4\delta + epoch\_duration - \Delta = GST + 2epoch\_duration + 4\delta - \Delta$. Hence, $t_s + \Delta - GST \leq 2epoch\_duration + 4\delta = O(f \cdot \delta)$.

Communication: *Communication complexity of* RARESYNC *is $O(n^2)$.*

Roughly speaking, every correct process will have entered e_{max} (or potentially $e_{final} = e_{max} + 1$) by time $GST + 2\delta$ (as seen in the proof of Result 6). From then on, it will enter at most one other epoch (e_{final}) before synchronizing (which is completed by time $t_s + \Delta$). As for the time interval $[GST, GST + 2\delta)$, due to $dissemination\_timer$'s interval of δ, a correct process can enter (at most) two other epochs during this period. Therefore, a correct process can enter (and send messages for) at most $O(1)$ epochs between GST and $t_s + \Delta$. The individual communication cost of a correct process is bounded by $O(n)$ words per epoch: $O(n)$ EPOCH- COMPLETED messages (each with a single word), and $O(n)$ ENTER- EPOCH messages (each with a single word, as a threshold signature counts as a single word). Thus, the communication complexity of RARESYNC is $O(1) \cdot O(n) \cdot n = O(n^2)$.

4.4 Formal proof

This section formally proves the correctness and establishes the complexity of RARESYNC (Algorithm 2). We start by defining the concept of a process' *behavior* and *timer history*.

Behaviors & timer histories. A *behavior* of a process P_i is a sequence of (1) message-sending events performed by P_i, (2) message-reception events performed by P_i, and (3) internal events performed by P_i (e.g., invocations of the $\mathsf{measure}(\cdot)$ and $\mathsf{cancel}()$ methods on the local timers). If an event e belongs to a behavior β_i, we write $e \in \beta_i$; otherwise, we write $e \notin \beta_i$. If an event e_1 precedes an event e_2 in a behavior β_i, we write $e_1 \overset{\beta_i}{\prec} e_2$. Note that, if $e_1 \overset{\beta_i}{\prec} e_2$ and e_1 occurs at some time t_1 and e_2 occurs at some time t_2, $t_1 \leq t_2$.

A *timer history* of a process P_i is a sequence of (1) invocations of the $\mathsf{measure}(\cdot)$ and $\mathsf{cancel}()$ methods on $view\_timer_i$ and $dissemination\_timer_i$, and (2) processed

[6] If $e_{max} = 1$, every correct process enters e_{max} by time GST.

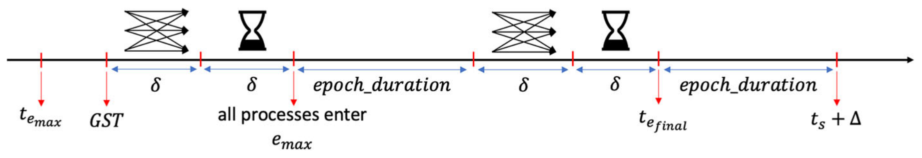

Fig. 3 Worst-case latency of RARESYNC: $t_s + \Delta - GST \leq 2epoch\_duration + 4\delta$

expiration events of $view\_timer_i$ and $dissemination\_timer_i$. Observe that a timer history of a process is a subsequence of the behavior of the process. We further denote by $h_i \mid_{view}$ the greatest subsequence of h_i associated with $view\_timer_i$, where h_i is a timer history of a process P_i. If an expiration event *Exp* of a timer is associated with an invocation *Inv* of the measure$(\cdot)$ method on the timer, we say that *Inv produces Exp*. Note that a single invocation of the measure$(\cdot)$ method can produce at most one expiration event.

Given an execution, we denote by β_i and h_i the behavior and the timer history of the process P_i, respectively.

4.4.1 Proof of correctness

In order to prove the correctness of RARESYNC, we need to prove that RARESYNC ensures the eventual synchronization property (Sect. 4.1).

We start by establishing some basic properties of Algorithm 2. These are encapsulated by several lemmas (specifically, Lemmas 1–8) which can be verified by simple visual code inspection. As such, we summarize them here but delegate their formal proofs to Appendix B.

First, notice that the value of $view_i$ variable at a correct process P_i is never smaller than 1 or greater than $f + 1$.

Lemma 1 *Let P_i be a correct process. Then, $1 \leq view_i \leq f + 1$ throughout the entire execution.*

It is also ensured that, if an invocation of the measure$(\cdot)$ method on $dissemination\_timer_i$ produces an expiration event, the expiration event immediately follows the invocation in the timer history h_i of a correct process P_i.

Lemma 2 *Let P_i be a correct process. Let Exp_d be any expiration event of $dissemination\_timer_i$ that belongs to h_i and let Inv_d be the invocation of the* measure$(\cdot)$ *method (on $dissemination\_timer_i$) that has produced Exp_d. Then, Exp_d immediately follows Inv_d in h_i.*

The next lemma shows that views entered by a correct process are monotonically increasing, as intended.

Lemma 3 *(Monotonically increasing views) Let P_i be a correct process. Let $e_1 =$ advance(v), $e_2 =$ advance(v') and $e_1 \overset{\beta_i}{\prec} e_2$. Then, $v' > v$.*

The next lemma shows that an invocation of the measure$(\cdot)$ method cannot be immediately followed by another invocation of the same method in a timer history (of a correct process) associated with $view\_timer_i$.

Lemma 4 *Let P_i be a correct process. Let Inv_v be any invocation of the* measure$(\cdot)$ *method on $view\_timer_i$ that belongs to h_i. Invocation Inv_v is not immediately followed by another invocation of the* measure$(\cdot)$ *method on $view\_timer_i$ in $h_i \mid_{view}$.*

As a direct consequence of Lemma 4, an expiration event of $view\_timer_i$ immediately follows (in a timer history associated with $view\_timer_i$) the measure$(\cdot)$ invocation that has produced it.

Lemma 5 *Let P_i be a correct process. Let Exp_v be any expiration event that belongs to $h_i \mid_{view}$ and let Inv_v be the invocation of the* measure$(\cdot)$ *method (on $view\_timer_i$) that has produced Exp_v. Then, Exp_v immediately follows Inv_v in $h_i \mid_{view}$.*

Consequently, the statement of Lemma 2 also holds for $view\_timer_i$:

Lemma 6 *Let P_i be a correct process. Let Exp_v be any expiration event of $view\_timer_i$ that belongs to h_i and let Inv_v be the invocation of the* measure$(\cdot)$ *method (on $view\_timer_i$) that has produced Exp_v. Then, Exp_v immediately follows Inv_v in h_i.*

Next, we show that the values of the $epoch_i$ and $view_i$ variables of a correct process P_i do not change between an invocation of the measure$(\cdot)$ method on $view\_timer_i$ and the processing of the expiration event the invocation produces.

Lemma 7 *Let P_i be a correct process. Let Inv_v denote an invocation of the* measure$(\cdot)$ *method on $view\_timer_i$ which produces an expiration event, and let Exp_v denote the expiration event produced by Inv_v. Let $epoch_i = e$ and $view_i = v$ when P_i invokes Inv_v. Then, when P_i processes Exp_v (line 4), $epoch_i = e$ and $view_i = v$.*

Finally, we show that correct processes cannot "jump" into an epoch, i.e., they must go into an epoch by going into its first view.

Lemma 8 *Let P_i be a correct process. Let* advance$(v) \in \beta_i$, *where v is the j-th view of an epoch e and $j > 1$. Then,* advance$(v - 1) \overset{\beta_i}{\prec}$ advance(v).

With the previous lemmas in place, the basic intended properties of Algorithm 2 are ensured. We now focus on the overarching properties of RARESYNC, such as the concept of entering an epoch.

We say that a correct process *enters* an epoch e at time t if and only if the process enters the first view of e (i.e., the view $(e-1)\cdot(f+1)+1$) at time t. Furthermore, a correct process *is in epoch* e between the time t (including t) at which it enters e and the time t' (excluding t') at which it enters (for the first time after entering e) another epoch e'. If another epoch is never entered, the process is in epoch e from time t onward. Recall that, by Lemma 3, a correct process enters each view at most once, which means that a correct process enters each epoch at most once.

The following lemma shows that, if a correct process broadcasts an EPOCH- COMPLETED message for an epoch (line 12), then the process has previously entered that epoch.

Lemma 9 *Let a correct process P_i send an* EPOCH-COMPLETED *message for an epoch e (line 12); let this sending event be denoted by e_{send}. Then,* $\mathsf{advance}(v) \overset{\beta_i}{\prec} e_{send}$*, where v is the first view of the epoch e.*

Proof At the moment of sending the message (line 12), the following holds: (1) $epoch_i = e$, and (2) $view_i = f+1$ (by the check at line 5 and Lemma 1). We denote by Inv_v the invocation of the $\mathsf{measure}(\cdot)$ method on $view\_timer_i$ producing the expiration event Exp_v leading to P_i broadcasting the EPOCH- COMPLETED message for e. Note that Inv_v precedes the sending of the EPOCH- COMPLETED message in β_i.

When processing Exp_v (line 4), the following was the state of P_i: $epoch_i = e$ and $view_i = f+1$. By Lemma 7, when P_i invokes Inv_v, $epoch_i = e$ and $view_i = f+1 > 1$. Therefore, Inv_v must have been invoked at line 8: Inv_v could not have been invoked either at line 2 or at line 29 since $view_i = f+1 \neq 1$ at that moment. Immediately after invoking Inv_v, P_i enters the $(f+1)$-st view of e (line 9), which implies that P_i enters the $(f+1)$-st view of e before it sends the EPOCH- COMPLETED message. Therefore, the lemma follows from Lemma 8. □

The next lemma shows that, if a correct process P_i updates its $epoch_i$ variable to $e > 1$, then (at least) $f+1$ correct processes have previously entered epoch $e-1$.

Lemma 10 *Let a correct process P_i update its $epoch_i$ variable to $e > 1$ at some time t. Then, at least $f+1$ correct processes have entered $e-1$ by time t.*

Proof Since P_i updates $epoch_i$ to $e > 1$ at time t, it does so at either:

- Line 15: In this case, P_i has received $2f+1$ EPOCH-COMPLETED messages for epoch $e-1$ (line 13), out of which (at least) $f+1$ were sent by correct processes.
- Line 21: In this case, P_i has received a threshold signature of epoch $e-1$ (line 19) built out of $2f+1$ partial signatures, out of which (at least) $f+1$ must have come from correct processes. Such a partial signature from a correct process can only be obtained by receiving an EPOCH-COMPLETED message for epoch $e-1$ from that process.

In both cases, $f+1$ correct processes have sent EPOCH-COMPLETED messages (line 12) for epoch $e-1$ by time t. By Lemma 9, all these correct processes have entered epoch $e-1$ by time t. □

Note that a correct process P_i *does not* enter an epoch immediately upon updating its $epoch_i$ variable, but only upon triggering the $\mathsf{advance}(\cdot)$ indication for the first view of that epoch (line 3 or line 30). We now prove that, if an epoch $e > 1$ is entered by a correct process at some time t, then epoch $e-1$ is entered by a (potentially different) correct process by time t.

Lemma 11 *Let a correct process P_i enter an epoch $e > 1$ at time t. Then, epoch $e-1$ was entered by a correct process by time t.*

Proof Since P_i enters $e > 1$ at time t (line 30), $epoch_i = e$ at time t. Hence, P_i has updated its $epoch_i$ variable to $e > 1$ by time t. Therefore, the lemma follows directly from Lemma 10. □

The next lemma shows that all epochs are eventually entered by some correct processes. In other words, correct processes keep transiting to new epochs forever.

Lemma 12 *Every epoch is eventually entered by a correct process.*

Proof Epoch 1 is entered by a correct process since every correct process initially triggers the $\mathsf{advance}(1)$ indication (line 3). Therefore, it is left to prove that all epochs greater than 1 are entered by a correct process. By contradiction, let $e+1$ be the smallest epoch not entered by a correct process, where $e \geq 1$.
Part 1. *No correct process P_i ever sets $epoch_i$ to an epoch greater than e.*
Since $e+1$ is the smallest epoch not entered by a correct process, no correct process ever enters any epoch greater than e (by Lemma 11). Furthermore, Lemma 10 shows that no correct process P_i ever updates its $epoch_i$ variable to an epoch greater than $e+1$.

Finally, P_i never sets $epoch_i$ to $e+1$ either. By contradiction, suppose that it does. In this case, P_i invokes the $\mathsf{measure}(\delta)$ method on $dissemination\_timer_i$ (either line 18 or line 24). Since P_i does not update $epoch_i$ to an epoch greater than $e+1$ (as shown in the previous paragraph), the previously invoked $\mathsf{measure}(\delta)$ method will never be canceled (neither at line 17 nor at line 23). This implies that $dissemination\_timer_i$ eventually expires (line 25), and P_i

enters epoch $e+1$ (line 30). Hence, a contradiction with the fact that epoch $e+1$ is never entered by a correct process.
Part 2. *Every correct process eventually enters epoch e.*
If $e = 1$, every correct process enters e as every correct process eventually executes line 3.

Let $e > 1$. Since $e > 1$ is entered by a correct process (line 30), the process has disseminated an ENTER- EPOCH message for e (line 26). This message is eventually received by every correct process since the network is reliable. If a correct process P_i has not previously set its $epoch_i$ variable to e, it does so upon the reception of the ENTER- EPOCH message (line 21). Hence, P_i eventually sets its $epoch_i$ variable to e.

Immediately after updating its $epoch_i$ variable to e (line 15 or line 21), P_i invokes measure(δ) on $dissemination\_timer_i$ (line 18 or line 24). Because P_i never updates $epoch_i$ to an epoch greater than e (by Part 1), $dissemination\_timer_i$ expires while $epoch_i = e$. When this happens (line 25), P_i enters epoch e (line 30). Thus, all correct processes eventually enter epoch e.
Epilogue. By Part 2, a correct process P_i eventually enters epoch e (line 3 or line 30); when P_i enters e, $epoch_i = e$ and $view_i = 1$. Moreover, just before entering e, P_i invokes the measure($\cdot$) method on $view\_timer_i$ (line 2 or line 29); let this invocation be denoted by Inv_v^1. As P_i never updates its $epoch_i$ variable to an epoch greater than e (by Part 1), Inv_v^1 eventually expires. When P_i processes the expiration of Inv_v^1 (line 4), $epoch_i = e$ and $view_i = 1 < f+1$ (by Lemma 7). Hence, P_i then invokes the measure($\cdot$) method on $view\_timer_i$ (line 8); when this occurs, $epoch_i = e$ and $view_i = 2$ (by line 6). Following the same argument as for Inv_v^1, $view\_timer_i$ expires for each view of epoch e.

Therefore, every correct process P_i eventually broadcasts an EPOCH- COMPLETED message for epoch e (line 12) when $view\_timer_i$ expires for the last view of epoch e. Thus, a correct process P_j eventually receives $2f+1$ EPOCH- COMPLETED messages for epoch e (line 13), and updates $epoch_j$ to $e+1$ (line 15). This contradicts Part 1, which implies that the lemma holds. □

We now introduce e_{final}, the first new epoch entered at or after GST.

Definition 2 We denote by e_{final} the smallest epoch such that the first correct process to enter e_{final} does so at time $t_{e_{final}} \geq GST$.

Note that e_{final} exists due to Lemma 12; recall that, by GST, an execution must be finite as no process is able to perform infinitely many steps in finite time. It is stated in Algorithm 1 that $view\_duration = \Delta + 2\delta$ (line 8). However, technically speaking, $view\_duration$ must be greater than $\Delta + 2\delta$ in order to not waste the "very last" moment of a $\Delta + 2\delta$ time period, i.e., we set $view\_duration = \Delta + 2\delta + \epsilon$, where ϵ is any positive constant. Therefore, in the rest of the section, we assume that $view\_duration = \Delta + 2\delta + \epsilon > \Delta + 2\delta$.

We now show that, if a correct process enters an epoch e at time $t_e \geq GST$ and sends an EPOCH- COMPLETED message for e, the EPOCH- COMPLETED message is sent at time $t_e + epoch\_duration$, where $epoch\_duration = (f+1) \cdot view\_duration$.

Lemma 13 *Let a correct process P_i enter an epoch e at time $t_e \geq GST$ and let P_i send an* EPOCH- COMPLETED *message for epoch e (line 12). The* EPOCH- COMPLETED *message is sent at time $t_e + epoch\_duration$.*

Proof We prove the lemma by backwards induction. Let t^* denote the time at which the EPOCH- COMPLETED message for epoch e is sent (line 12).
Base step: *The $(f+1)$-st view of the epoch e is entered by P_i at time t^{f+1} such that $t^* - t^{f+1} = 1 \cdot view\_duration$.*
When sending the EPOCH- COMPLETED message (line 12), the following holds: $epoch_i = e$ and $view_i = f+1$ (due to the check at line 5 and Lemma 1). Let Exp_v^{f+1} denote the expiration event of $view\_timer_i$ processed just before broadcasting the message (line 4). When processing Exp_v^{f+1}, we have that $epoch_i = e$ and $view_i = f+1$. When P_i has invoked Inv_v^{f+1}, where Inv_v^{f+1} is the invocation of the measure($\cdot$) method which has produced Exp_v^{f+1}, we have that $epoch_i = e$ and $view_i = f+1$ (by Lemma 7). As $f+1 \neq 1$, Inv_v^{f+1} is invoked at line 8 at some time $t^{f+1} \leq t^*$. Finally, P_i enters the $(f+1)$-st view of the epoch e at line 9 at time t^{f+1}. By Lemma 8, we have that $t^{f+1} \geq t_e \geq GST$. As local clocks do not drift after GST, we have that $t^* - t^{f+1} = view\_duration$ (due to line 8), which concludes the base step.
Induction step: *Let $j \in [1, f]$. The j-th view of the epoch e is entered by P_i at time t^j such that $t^* - t^j = (f+2-j) \cdot view\_duration$.*
Induction hypothesis: For every $k \in [j+1, f+1]$, the k-th view of the epoch e is entered by P_i at time t^k such that $t^* - t^k = (f+2-k) \cdot view\_duration$.
Let us consider the $(j+1)$-st view of the epoch e; note that $j+1 \neq 1$. Hence, the $(j+1)$-st view of the epoch e is entered by P_i at some time t^{j+1} at line 9, where $t^* - t^{j+1} = (f+2-j-1) \cdot view\_duration = (f+1-j) \cdot view\_duration$ (by the induction hypothesis). Let Exp_v^j denote the expiration event of $view\_timer_i$ processed at time t^{j+1} (line 4). When processing Exp_v^j, we have that $epoch_i = e$ and $view_i = j$ (due to line 6). When P_i has invoked Inv_v^j at some time t^j, where Inv_v^j is the invocation of the measure($\cdot$) method which has produced Exp_v^j, we have that $epoch_i = e$ and $view_i = j$ (by Lemma 7). Inv_v^j could have been invoked either at line 2, or at line 8, or at line 29:

- Line 2: In this case, P_i enters the j-th view of the epoch e at time t^j at line 3, where $j = 1$ (by line 3). Moreover, we have that $t^j \geq GST$ as $t^j = t_e$ (by definition). As local clocks do not drift after GST, we have that $t^{j+1} - t^j = view\_duration$, which implies that $t^* - t^j = t^* - t^{j+1} + view\_duration = (f + 1 - j + 1) \cdot view\_duration = (f + 2 - j) \cdot view\_duration$. Hence, in this case, the induction step is concluded.

- Line 8: P_i enters the j-th view of the epoch e at line 9 at time t^j, where $j > 1$ (by Lemma 1 and line 6). By Lemmas 3 and 8, we have that $t^j \geq t_e \geq GST$. As local clocks do not drift after GST, we have that $t^{j+1} - t^j = view\_duration$, which implies that $t^* - t^j = (f + 2 - j) \cdot view\_duration$. Hence, the induction step is concluded even in this case.
- Line 29: In this case, P_i enters the j-th view of the epoch e at time t^j at line 30, where $j = 1$ as $view_i = 1$ (by line 27). Moreover, $t^j = t_e \geq GST$ (by definition). As local clocks do not drift after GST, we have that $t^{j+1} - t^j = view\_duration$, which implies that $t^* - t^j = t^* - t^{j+1} + view\_duration = (f+1-j+1) \cdot view\_duration = (f + 2 - j) \cdot view\_duration$. Hence, even in this case, the induction step is concluded.

As the induction step is concluded in all possible scenarios, the backwards induction holds. Therefore, P_i enters the first view of the epoch e (and, thus, the epoch e) at time t_e (recall that the first view of any epoch is entered at most once by Lemma 3) such that $t^* - t_e = (f + 1) \cdot view\_duration = epoch\_duration$, which concludes the proof. □

The following lemma shows that no correct process broadcasts an EPOCH- COMPLETED message for an epoch $\geq e_{final}$ before time $t_{e_{final}} + epoch\_duration$.

Lemma 14 *No correct process broadcasts an* EPOCH- COMPLETED *message for an epoch $e' \geq e_{final}$ (line 12) before time $t_{e_{final}} + epoch\_duration$.*

Proof Let t^* be the first time a correct process, denoted by P_i, sends an EPOCH- COMPLETED message for an epoch $e' \geq e_{final}$ (line 12); if t^* is not defined, the lemma trivially holds. By Lemma 9, P_i has entered epoch e' at some time $t_{e'} \leq t^*$. If $e' = e_{final}$, then $t_{e'} \geq t_{e_{final}} \geq GST$. If $e' > e_{final}$, by Lemma 11, $t_{e'} \geq t_{e_{final}} \geq GST$. Therefore, $t^* = t_{e'} + epoch\_duration$ (by Lemma 13), which means that $t^* \geq t_{e_{final}} + epoch\_duration$. □

Next, we show during which periods a correct process is in which view of the epoch e_{final}.

Lemma 15 *Consider a correct process P_i.*

- *For any $j \in [1, f]$, P_i enters the j-th view of the epoch e_{final} at some time t^j, where $t^j \in \big[t_{e_{final}} + (j - 1) \cdot view\_duration, t_{e_{final}} + (j-1) \cdot view\_duration + 2\delta\big]$, and stays in the view until (at least) time $t^j + view\_duration$ (excluding time $t^j + view\_duration$).*
- *For $j = f + 1$, P_i enters the j-th view of the epoch e_{final} at some time t^j, where $t^j \in \big[t_{e_{final}} + f \cdot view\_duration, t_{e_{final}} + f \cdot view\_duration + 2\delta\big]$, and stays in the view until (at least) time $t_{e_{final}} + epoch\_duration$ (excluding time $t_{e_{final}} + epoch\_duration$).*

Proof Note that no correct process broadcasts an EPOCH- COMPLETED message for an epoch $\geq e_{final}$ (line 12) before time $t_{e_{final}} + epoch\_duration$ (by Lemma 14). We prove the lemma by induction.

Base step: *The statement of the lemma holds for $j = 1$.*
If $e_{final} > 1$, every correct process receives an ENTER- EPOCH message (line 19) for epoch e_{final} by time $t_{e_{final}} + \delta$ (since $t_{e_{final}} \geq GST$). As no correct process broadcasts an EPOCH- COMPLETED message for an epoch $\geq e_{final}$ before time $t_{e_{final}} + epoch\_duration > t_{e_{final}} + \delta$, P_i sets its $epoch_i$ variable to e_{final} (line 21) and invokes the measure(δ) method on $dissemination\_timer_i$ (line 24) by time $t_{e_{final}} + \delta$. Because of the same reason, the $dissemination\_timer_i$ expires by time $t_{e_{final}} + 2\delta$ (line 25); at this point in time, $epoch_i = e_{final}$. Hence, P_i enters the first view of e_{final} by time $t_{e_{final}} + 2\delta$ (line 30). Observe that, if $e_{final} = 1$, P_i enters e_{final} at time $t_{e_{final}}$ (as every correct process starts executing Algorithm 2 at $GST = t_{e_{final}}$). Thus, $t^1 \in [t_{e_{final}}, t_{e_{final}} + 2\delta]$.

Prior to entering the first view of e_{final}, P_i invokes the measure($view\_duration$) method on $view\_timer_i$ (line 2 or line 29); we denote this invocation by Inv_v. By Lemma 14, Inv_v cannot be canceled (line 16 or line 22) as $t_{e_{final}} + epoch\_duration > t_{e_{final}} + 2\delta + view\_duration$. Therefore, Inv_v produces an expiration event Exp_v which is processed by P_i at time $t^1 + view\_duration$ (since $t^1 \geq GST$ and local clocks do not drift after GST).

Let us investigate the first time P_i enters another view after entering the first view of e_{final}. This could happen at the following places of Algorithm 2:

- Line 9: By Lemma 6, we conclude that this occurs at time $t^* \geq t^1 + view\_duration$. Therefore, in this case, P_i is in the first view of e_{final} during the time period $[t^1, t^1 + view\_duration)$. The base step is proven in this case.
- Line 30: By contradiction, suppose that this happens before time $t^1 + view\_duration$. Hence, the measure($\cdot$) method was invoked on $dissemination\_timer_i$ (line 18 or line 24) before time $t^1 + view\_duration$ and after the invocation of Inv_v (by Lemma 2). Thus, Inv_v is canceled (line 16 or line 22), which is impossible (as previously proven). Hence, P_i is in the first view of e_{final} during (at least) the time period

$[t^1, t^1 + view\_duration)$, which implies that the base step is proven even in this case.

Induction step: *The statement of the lemma holds for j, where $1 < j \leq f + 1$.*
Induction hypothesis: The statement of the lemma holds for every $k \in [1, j - 1]$.
Consider the $(j - 1)$-st view of e_{final} denoted by v_{j-1}. Recall that t^{j-1} denotes the time at which P_i enters v_{j-1}. Just prior to entering v_{j-1} (line 3 or line 9 or line 30), P_i has invoked the $\mathsf{measure}(view\_duration)$ method on $view\_timer_i$ (line 2 or line 8 or line 29); let this invocation be denoted by Inv_v. When P_i invokes Inv_v, we have that $epoch_i = e_{final}$ and $view_i = j - 1$. As in the base step, Lemma 14 shows that Inv_v cannot be canceled (line 16 or line 22) as $t_{e_{final}} + epoch\_duration > t^{j-1} + view\_duration$ since $t^{j-1} \leq t_{e_{final}} + (j - 2) \cdot view\_duration + 2\delta$ (by the induction hypothesis). We denote by Exp_v the expiration event produced by Inv_v. By Lemma 7, when P_i processes Exp_v (line 4), we have that $epoch_i = e_{final}$ and $view_i = j - 1 < f + 1$. Hence, P_i enters the j-th view of e_{final} at time $t^j = t^{j-1} + view\_duration$ (line 9), which means that $t^j \in \big[t_{e_{final}} + (j - 1) \cdot view\_duration, t_{e_{final}} + (j - 1) \cdot view\_duration + 2\delta\big]$.

We now separate two cases:

- Let $j < f + 1$. Just prior to entering the j-th view of e_{final} (line 9), P_i invokes the $\mathsf{measure}(view\_duration)$ method on $view\_timer_i$ (line 8); we denote this invocation by Inv'_v. By Lemma 14, Inv'_v cannot be canceled (line 16 or line 22) as $t_{e_{final}} + epoch\_duration > t_{e_{final}} + (j - 1) \cdot view\_duration + 2\delta + view\_duration$. Therefore, Inv'_v produces an expiration event Exp'_v which is processed by P_i at time $t^j + view\_duration$ (since $t^j \geq GST$ and local clocks do not drift after GST). Let us investigate the first time P_i enters another view after entering the j-th view of e_{final}. This could happen at the following places of Algorithm 2:
  - Line 9: By Lemma 6, we conclude that this occurs at time $\geq t^j + view\_duration$. Therefore, in this case, P_i is in the j-th view of e_{final} during the time period $[t^j, t^j + view\_duration)$. The induction step is proven in this case.
  - Line 30: By contradiction, suppose that this happens before time $t^j + view\_duration$. Hence, the $\mathsf{measure}(\cdot)$ method was invoked on $dissemination\_timer_i$ (line 18 or line 24) before time $t^j + view\_duration$ and after the invocation of Inv'_v (by Lemma 2). Thus, Inv'_v is canceled (line 16 or line 22), which is impossible (as previously proven). Hence, P_i is in the j-th view of e_{final} during (at least) the time period $[t^j, t^j + view\_duration)$, which concludes the induction step even in this case.
- Let $j = f + 1$. Just prior to entering the j-th view of e_{final} (line 9), P_i invokes the $\mathsf{measure}(view\_duration)$ method on $view\_timer_i$ (line 8); we denote this invocation by Inv'_v. When Inv'_v was invoked, $epoch_i = e_{final}$ and $view_i = f + 1$. By Lemma 14, we know that the earliest time Inv'_v can be canceled (line 16 or line 22) is $t_{e_{final}} + epoch\_duration$. Let us investigate the first time P_i enters another view after entering the j-th view of e_{final}. This could happen at the following places of Algorithm 2:
  - Line 9: This means that, when processing the expiration event of $view\_timer_i$ (denoted by Exp^*_v) at line 4 (before executing the check at line 5), $view_i < f + 1$. Hence, Exp^*_v is not produced by Inv'_v (by Lemma 7). By contradiction, suppose that Exp^*_v is processed before time $t_{e_{final}} + epoch\_duration$. In this case, Exp^*_v is processed before the expiration event produced by Inv'_v would (potentially) be processed (which is $t_{e_{final}} + epoch\_duration$ at the earliest). Thus, Inv'_v must be immediately followed by an invocation of the $\mathsf{cancel}()$ method on $view\_timer_i$ in $h_i|_{view}$ (by Lemmas 4 and 5). As previously shown, the earliest time Inv'_v can be canceled is $t_{e_{final}} + epoch\_duration$, which implies that Exp^*_v cannot be processed before time $t_{e_{final}} + epoch\_duration$. Therefore, Exp^*_v is processed at $t_{e_{final}} + epoch\_duration$ (at the earliest), which concludes the induction step for this case.
  - Line 30: Suppose that, by contradiction, this happens before time $t_{e_{final}} + epoch\_duration$. Hence, the $\mathsf{measure}(\cdot)$ method was invoked on $dissemination\_timer_i$ (line 18 or line 24) before time $t_{e_{final}} + epoch\_duration$ (by Lemma 2) and after P_i has entered the j-th view of e_{final}, which implies that Inv'_v is canceled before time $t_{e_{final}} + epoch\_duration$ (line 16 or line 22). However, this is impossible as the earliest time for Inv'_v to be canceled is $t_{e_{final}} + epoch\_duration$. Hence, P_i enters another view at time $t_{e_{final}} + epoch\_duration$ (at the earliest), which concludes the induction step in this case.

The conclusion of the induction step concludes the proof of the lemma. □

Finally, we prove that RareSync ensures the eventual synchronization property.

Theorem 1 *(Eventual synchronization)* RareSync *ensures eventual synchronization. Moreover, the first synchronization time at or after GST occurs by time $t_{e_{final}} + f \cdot view\_duration + 2\delta$.*

Proof Lemma 15 proves that all correct processes overlap in each view of e_{final} for (at least) Δ time. As the leader of one view of e_{final} must be correct (since $\mathsf{leader}(\cdot)$ is a round-robin function), the eventual synchronization is satisfied by RARESYNC: correct processes synchronize in (at least) one of the views of e_{final}. Finally, as the last view of e_{final} is entered by every correct process by time $t^* = t_{e_{final}} + f \cdot view\_duration + 2\delta$ (by Lemma 15), the first synchronization time at or after GST must occur by time t^*. □

4.4.2 Proof of complexity

We start by showing that, if a correct process sends an EPOCH-COMPLETED message for an epoch e, then the "most recent" epoch entered by the process is e.

Lemma 16 *Let P_i be a correct process and let P_i send an* EPOCH- COMPLETED *message for an epoch e (line 12). Then, e is the last epoch entered by P_i in β_i before sending the* EPOCH- COMPLETED *message.*

Proof By Lemma 9, P_i enters e before sending the EPOCH-COMPLETED message for e. By contradiction, suppose that P_i enters some other epoch e^* after entering e and before sending the EPOCH- COMPLETED message for e. By Lemma 3, $e^* > e$.

When P_i enters e^* (line 30), $epoch_i = e^*$. As the value of the $epoch_i$ variable only increases throughout the execution, P_i does not send the EPOCH- COMPLETED message for e after entering $e^* > e$. Thus, we reach a contradiction, and the lemma holds. □

Next, we show that, if a correct process sends an ENTER-EPOCH message for an epoch e at time t, the process enters e at time t.

Lemma 17 *Let a correct process P_i send an* ENTER- EPOCH *message (line 26) for an epoch e at time t. Then, P_i enters e at time t.*

Proof When P_i sends the ENTER- EPOCH message, we have that $epoch_i = e$. Hence, P_i enters e at time t (line 30). □

Next, we show that a correct process sends (at most) $O(n)$ EPOCH- COMPLETED messages for a specific epoch e.

Lemma 18 *For any epoch e and any correct process P_i, P_i sends at most $O(n)$* EPOCH- COMPLETED *messages for e (line 12).*

Proof Let Exp_v denote the first expiration event of $view\_timer_i$ which P_i processes (line 4) in order to broadcast the EPOCH-COMPLETED message for e (line 12); if Exp_v does not exist, the lemma trivially holds. Hence, let Exp_v exist.

When Exp_v was processed, $epoch_i = e$. Let Inv'_v denote the first invocation of the $\mathsf{measure}(\cdot)$ method on $view\_timer_i$ after the processing of Exp_v. If Inv'_v does not exist, there does not exist an expiration event of $view\_timer_i$ processed after Exp_v (by Lemma 6), which implies that the lemma trivially holds.

Let us investigate where Inv'_v could have been invoked:

- Line 8: By Lemma 6, we conclude that the processing of Exp_v leads to Inv'_v. However, this is impossible as the processing of Exp_v leads to the broadcasting of the EPOCH- COMPLETED messages (see the check at line 5).
- Line 29: In this case, P_i processes an expiration event Exp_d of $dissemination\_timer_i$ (line 25). By Lemma 2, the invocation Inv_d of the $\mathsf{measure}(\cdot)$ method on $dissemination\_timer_i$ immediately precedes Exp_d in h_i. Hence, Inv_d follows Exp_v in h_i and Inv_d could have been invoked either at line 18 or at line 24. Just before invoking Inv_d, P_i changes its $epoch_i$ variable to a value greater than e (line 15 or line 21; the value of $epoch_i$ only increases throughout the execution).

Therefore, when Inv'_v is invoked, $epoch_i > e$. As the value of the $epoch_i$ variable only increases throughout the execution, P_i broadcasts the EPOCH- COMPLETED messages for e at most once (by Lemma 6), which concludes the proof. □

The following lemma shows that a correct process sends (at most) $O(n)$ ENTER- EPOCH messages for a specific epoch e.

Lemma 19 *For any epoch e and any correct process P_i, P_i sends at most $O(n)$* ENTER- EPOCH *messages for e (line 26).*

Proof Let Exp_d denote the first expiration event of $dissemination\_timer_i$ which P_i processes (line 25) in order to broadcast the ENTER- EPOCH message for e (line 26); if Exp_d does not exist, the lemma trivially holds. When Exp_d was processed, $epoch_i = e$. Let Inv'_d denote the first invocation of the $\mathsf{measure}(\cdot)$ method on $dissemination\_timer_i$ after the processing of Exp_d. If Inv'_d does not exist, there does not exist an expiration event of $dissemination\_timer_i$ processed after Exp_d (by Lemma 2), which implies that the lemma trivially holds.

Inv'_d could have been invoked either at line 18 or at line 24. However, before that (still after the processing of Exp_d), P_i changes its $epoch_i$ variable to a value greater than e (line 15 or line 21). Therefore, when Inv'_d is invoked, $epoch_i > e$. As the value of the $epoch_i$ variable only increases throughout the execution, P_i broadcasts the ENTER- EPOCH messages for e at most once (by Lemma 2), which concludes the proof. □

Next, we show that, after GST, two "epoch-entering" events are separated by at least δ time.

Lemma 20 *Let P_i be a correct process. Let P_i trigger* $\mathsf{advance}(v)$ *at time $t \geq GST$ and let P_i trigger* $\mathsf{advance}(v')$

at time t' such that (1) $\mathsf{advance}(v) \overset{\beta_i}{\prec} \mathsf{advance}(v')$*, and (2) v (resp., v') is the first view of an epoch e (resp., e'). Then, $t' \geq t + \delta$.*

Proof Let $\mathsf{advance}(v^*)$, where v^* is the first view of an epoch e^*, be the first "epoch-entering" event following $\mathsf{advance}(v)$ in β_i (i.e., $\mathsf{advance}(v) \overset{\beta_i}{\prec} \mathsf{advance}(v^*)$); let $\mathsf{advance}(v^*)$ be triggered at time t^*. In order to prove the lemma, it suffices to show that $t^* \geq t + \delta$.

The $\mathsf{advance}(v^*)$ upcall is triggered at line 30. Let Exp_d denote the processed expiration event of $dissemination\_timer_i$ (line 25) which leads P_i to trigger $\mathsf{advance}(v^*)$. Let Inv_d denote the invocation of the $\mathsf{measure}(\delta)$ on $dissemination\_timer_i$ that has produces Exp_d. By Lemma 2, Inv_d immediately precedes Exp_d in the timer history h_i of P_i. Note that Inv_d was invoked after P_i has entered e (this follows from Lemma 2 and the fact that P_i enters e after invoking $\mathsf{measure}(\cdot)$ on $view\_timer_i$), which means that Inv_d was invoked at some time $\geq t \geq GST$. As local clocks do not drift after GST, Exp_d is processed at some time $\geq t + \delta$, which concludes the proof. □

Next, we define t_s as the first synchronization time at or after GST.

Definition 3 We denote by t_s the first synchronization time at or after GST (i.e., $t_s \geq GST$).

The next lemma shows that no correct process enters any epoch greater than e_{final} by $t_s + \Delta$. This lemma is the consequence of Lemma 14 and Theorem 1.

Lemma 21 *No correct process enters an epoch greater than e_{final} by time $t_s + \Delta$.*

Proof By Lemma 14, no correct process enters an epoch $> e_{final}$ before time $t_{e_{final}} + epoch\_duration$. By Theorem 1, we have that $t_s < t_{e_{final}} + epoch\_duration - \Delta$, which implies that $t_{e_{final}} + epoch\_duration > t_s + \Delta$. Hence, the lemma. □

Next, we define e_{max} as the greatest epoch entered by a correct process before time GST. Note that e_{max} is properly defined in any execution as only finite executions are possible until GST.

Definition 4 We denote by e_{max} the greatest epoch entered by a correct process before GST. If no such epoch exists, $e_{max} = 0$.

The next lemma shows that e_{final} (Definition 2) is $e_{max} + 1$.

Lemma 22 $e_{final} = e_{max} + 1$.

Proof If $e_{max} = 0$, then $e_{final} = 1$. Hence, let $e_{max} > 0$ in the rest of the proof.

By the definitions of e_{final} (Definition 2) and e_{max} (Definition 4) and by Lemma 11, $e_{final} \geq e_{max} + 1$. Therefore, we need to prove that $e_{final} \leq e_{max} + 1$.

By contradiction, suppose that $e_{final} > e_{max} + 1$. By Lemma 11, epoch $e_{final} - 1$ was entered by the first correct process at some time $t_{prev} \leq t_{e_{final}}$. Note that $e_{final} - 1 \geq e_{max} + 1$. Moreover, $t_{prev} \geq GST$; otherwise, we would contradict the definition of e_{max}. Thus, the first new epoch to be entered by a correct process at or after GST is not e_{final}, i.e., we contradict Definition 2. Hence, the lemma holds. □

Next, we show that every correct process enters epoch e_{max} by time $GST + 2\delta$ or epoch $e_{final} = e_{max} + 1$ by time $GST + 3\delta$.

Lemma 23 *Every correct process (1) enters epoch e_{max} by $GST + 2\delta$, or (2) enters epoch $e_{max} + 1$ by $GST + 3\delta$.*

Proof Lemma 22 shows that e_{final} is $e_{max} + 1$. Recall that $t_{e_{final}} \geq GST$. Consider a correct process P_i. If $e_{max} = 1$ (resp., $e_{final} = 1$), then P_i enters e_{max} (resp., e_{final}) by time GST, which concludes the lemma. Hence, let $e_{max} > 1$; thus, $e_{final} > 1$ by Lemma 22.

Lemma 14 proves that no correct process broadcasts an EPOCH- COMPLETED message for an epoch $\geq e_{max} + 1$ before time $t_{e_{final}} + epoch\_duration \geq GST + epoch\_duration$.

By time $GST + \delta$, every correct process P_i receives an ENTER- EPOCH message for epoch $e_{max} > 1$ (line 19) sent by the correct process which has entered e_{max} before GST (the message is sent at line 26). Therefore, by time $GST + \delta$, $epoch_i$ is either e_{max} or $e_{max} + 1$; note that $epoch_i$ cannot take a value greater than $e_{max} + 1$ before time $GST + epoch\_duration > GST + \delta$ since no correct process broadcasts an EPOCH- COMPLETED message for an epoch $\geq e_{max} + 1$ before this time.

Let us consider both scenarios:

- Let $epoch_i = e_{max} + 1$ by time $GST + \delta$. In this case, $dissemination\_timer_i$ expires in δ time (line 25), and P_i enters $e_{max} + 1$ by time $GST + 2\delta$ (line 30) as $GST + epoch\_duration > GST + 2\delta$. Hence, the statement of the lemma is satisfied in this case.
- Let $epoch_i = e_{max}$ by time $GST + \delta$. If, within δ time from updating $epoch_i$ to e_{max}, P_i does not cancel its $dissemination\_timer_i$, $dissemination\_timer_i$ expires (line 4), and P_i enters e_{max} by time $GST + 2\delta$. Otherwise, $epoch_i = e_{max} + 1$ by time $GST + 2\delta$ as $dissemination\_timer_i$ was canceled; $epoch_i$ cannot take any other value as EPOCH- COMPLETED messages are not broadcast before time $GST + epoch\_duration > GST + 2\delta$. As in the previous case, $dissemination\_timer_i$ expires in δ time (line 25), and P_i enters $e_{max} + 1$ by time $GST + 3\delta$ (line 30) as $GST + epoch\_duration > GST + 3\delta$. Hence, the statement of the lemma holds in this case, as well.

Since the lemma is satisfied in both possible scenarios, the proof is concluded. □

The direct consequence of Lemma 22 is that $t_{e_{final}} \leq GST + epoch\_duration + 4\delta$.

Lemma 24 $t_{e_{final}} \leq GST + epoch\_duration + 4\delta$.

Proof By contradiction, let $t_{e_{final}} > GST + epoch\_duration + 4\delta$. Lemma 23 proves that every correct process enters epoch e_{max} by time $GST + 2\delta$ or epoch $e_{final} = e_{max} + 1$ by time $GST + 3\delta$. Additionally, Lemma 14 proves that no correct process broadcasts an EPOCH- COMPLETED message for an epoch $\geq e_{final}$ (line 12) before time $t_{e_{final}} + epoch\_duration > GST + 2 \cdot epoch\_duration + 4\delta$.

If any correct process enters $e_{max} + 1$ by time $GST + 3\delta$, we reach a contradiction with the fact that $t_{e_{final}} > GST + epoch\_duration + 4\delta$ since $e_{final} = e_{max} + 1$ (by Lemma 22). Therefore, all correct processes enter e_{max} by time $GST + 2\delta$.

Since $t_{e_{final}} > GST + epoch\_duration + 4\delta$, no correct process P_i updates its $epoch_i$ variable to $e_{max} + 1$ (at line 15 or line 21) by time $GST + epoch\_duration + 3\delta$ (otherwise, P_i would have entered $e_{max} + 1$ by time $GST + epoch\_duration + 4\delta$, which contradicts $t_{e_{final}} > GST + epoch\_duration + 4\delta$). By time $GST + epoch\_duration + 2\delta$, all correct processes broadcast an EPOCH- COMPLETED message for e_{max} (line 12). By time $GST + epoch\_duration + 3\delta$, every correct process P_i receives $2f + 1$ EPOCH- COMPLETED messages for e_{max} (line 13), and updates its $epoch_i$ variable to $e_{max} + 1$ (line 15). This represents a contradiction with the fact that P_i does not update its $epoch_i$ variable to $e_{max} + 1$ by time $GST + epoch\_duration + 3\delta$, which concludes the proof. □

The final lemma shows that no correct process enters more than $O(1)$ epochs during the time period $[GST, t_s + \Delta]$.

Lemma 25 *No correct process enters more than $O(1)$ epochs in the time period $[GST, t_s + \Delta]$.*

Proof Consider a correct process P_i. Process P_i enters epoch e_{max} by time $GST + 2\delta$ or P_i enters epoch $e_{max} + 1$ by time $GST + 3\delta$ (by Lemma 23). Lemma 22 shows that $e_{final} = e_{max} + 1$. Finally, no correct process enters an epoch greater than $e_{final} = e_{max} + 1$ by time $t_s + \Delta$ (by Lemma 21).

Let us consider two scenarios according to Lemma 23:

1. By time $GST + 2\delta$, P_i enters e_{max}; let P_i enter e_{max} at time $t^* \leq GST + 2\delta$. By Lemma 3, during the time period $[t^*, t_s + \Delta]$, P_i enters (at most) $2 = O(1)$ epochs (epochs e_{max} and $e_{max} + 1$). Finally, during the time period $[GST, t^*)$, Lemma 20 shows that P_i enters (at most) $2 = O(1)$ epochs (as $t^* \leq GST + 2\delta$). Hence, in this case, P_i enters (at most) $4 = O(1)$ epochs during the time period $[GST, t_s + \Delta]$.
2. By time $GST + 3\delta$, P_i enters $e_{max} + 1$; let P_i enter $e_{max} + 1$ at time $t^* \leq GST + 3\delta$. By Lemma 3, during the time period $[t^*, t_s + \Delta]$, P_i enters (at most) $1 = O(1)$ epoch (epoch $e_{max} + 1$). Finally, during the time period $[GST, t^*)$, Lemma 20 shows that P_i enters (at most) $3 = O(1)$ epochs (as $t^* \leq GST + 3\delta$). Hence, in this case, P_i enters (at most) $4 = O(1)$ epochs during the time period $[GST, t_s + \Delta]$.

Hence, during the time period $[GST, t_s + \Delta]$, P_i enters (at most) $4 = O(1)$ epochs. □

Finally, we prove that RARESYNC achieves $O(n^2)$ communication and $O(f \cdot \delta)$ latency.

Theorem 2 *(Complexity)* RARESYNC *achieves $O(n^2)$ communication complexity and $O(f \cdot \delta)$ latency complexity.*

Proof Fix a correct process P_i. For every epoch e, P_i sends (at most) $O(n)$ EPOCH- COMPLETED and ENTER- EPOCH messages for e (by Lemmas 18 and 19). Moreover, if P_i sends an EPOCH- COMPLETED message for an epoch e at time t, then e is the last epoch entered by P_i prior to sending the message (by Lemma 16). □

5 SQUAD

This section introduces SQUAD, a partially synchronous Byzantine consensus protocol with optimal resilience [9]. SQUAD simultaneously achieves (1) $O(n^2)$ communication complexity, matching the Dolev-Reischuk bound [6], and (2) $O(f \cdot \delta)$ latency complexity, matching the Dolev-Strong bound [17].

First, we present QUAD, a partially synchronous Byzantine consensus protocol ensuring weak validity (Sect. 5.1). QUAD achieves quadratic communication complexity and linear latency complexity. We provide a short overview of the proof of QUAD's correctness and complexity, followed by the complete formal proof (Sect. 5.2). Then, we construct SQUAD by adding a simple preprocessing phase to QUAD (Sect. 5.3), which we then formally prove (Sect. 5.4).

5.1 QUAD

QUAD is a partially synchronous Byzantine consensus protocol satisfying the weak validity property:

- *Weak validity:* If all processes are correct, then a value decided by a process was proposed.

QUAD achieves (1) quadratic communication complexity, and (2) linear latency complexity. Interestingly, the Dolev-Reischuk lower bound [6] does not apply to Byzantine protocols satisfying weak validity; hence, we do not know whether QUAD has optimal communication complexity. As explained in Sect. 5.3, we accompany QUAD by a preprocessing phase to obtain SQUAD.

QUAD (Algorithm 3) uses the same view core module as HotStuff [13], i.e., the view logic of QUAD is identical to that of HotStuff. Moreover, QUAD uses RARESYNC as its view synchronizer, achieving synchronization with $O(n^2)$ communication. The combination of HotStuff's view core and RARESYNC ensures that each correct process sends $O(n)$ words after GST (and before the decision), i.e., $C = O(n)$ in QUAD. Following the formula introduced in Sect. 1, QUAD indeed achieves $n \cdot C + S = n \cdot O(n) + O(n^2) = O(n^2)$ communication complexity. Due to the linear latency of RARESYNC, QUAD also achieves $O(f \cdot \delta)$ latency complexity.

5.1.1 View core

We now give a brief description of the view core module of QUAD. The complete pseudocode of this module can be found in Sect. 5.2 (and in [13]).

Each correct process keeps track of two critical variables: (1) the *prepare* quorum certificate (QC), and (2) the *locked* QC. Each of these represents a process' estimation of the value that will be decided, although with a different degree of certainty. For example, if a correct process decides a value v, it is guaranteed that (at least) $f + 1$ correct processes have v in their locked QC. Moreover, it is ensured that no correct process updates (from this point onward) its prepare or locked QC to any other value, thus ensuring agreement. Lastly, a QC is a (constant-sized) threshold signature.

The structure of a view follows the "all-to-leader, leader-to-all" communication pattern. Specifically, each view is comprised of the following four phases:

1. **Prepare:** A process sends to the leader a VIEW- CHANGE message containing its prepare QC. Once the leader receives $2f + 1$ VIEW- CHANGE messages, it selects the prepare QC from the "latest" view. The leader sends this QC to all processes via a PREPARE message. Once a process receives the PREPARE message from the leader, it supports the received prepare QC if (1) the received QC is consistent with its locked QC, or (2) the received QC is "more recent" than its locked QC. If the process supports the received QC, it acknowledges this by sending a PREPARE- VOTE message to the leader.
2. **Precommit:** Once the leader receives $2f + 1$ PREPARE- VOTE messages, it combines them into a cryptographic proof σ that "enough" processes have supported its "prepare-phase" value; σ is a threshold signature. Then, it disseminates σ to all processes via a PRECOMMIT message. Once a process receives the PRECOMMIT message carrying σ, it updates its prepare QC to σ and sends back to the leader a PRECOMMIT- VOTE message.
3. **Commit:** Once the leader receives $2f + 1$ PRECOMMIT- VOTE messages, it combines them into a cryptographic proof σ' that "enough" processes have adopted its "precommit-phase" value (by updating their prepare QC); σ' is a threshold signature. Then, it disseminates σ' to all processes via a COMMIT message. Once a process receives the COMMIT message carrying σ', it updates its locked QC to σ' and sends back to the leader a COMMIT- VOTE message.
4. **Decide:** Once the leader receives $2f + 1$ COMMIT- VOTE messages, it combines them into a threshold signature σ'', and relays σ'' to all processes via a DECIDE message. When a process receives the DECIDE message carrying σ'', it decides the value associated with σ''.

As a consequence of the "all-to-leader, leader-to-all" communication pattern and the constant size of messages, the leader of a view sends $O(n)$ words, while a non-leader process sends $O(1)$ words.

The view core module provides the following interface:

- **Request** start_executing(View v): The view core starts executing the logic of view v and abandons the previous view. Concretely, it stops accepting and sending messages for the previous view, and it starts accepting, sending, and replying to messages for view v. The state of the view core is kept across views (e.g., the prepare and locked QCs).
- **Indication** decide(Value *decision*): The view core decides value *decision* (this indication is triggered at most once).

5.1.2 Protocol description

The protocol (Algorithm 3) amounts to a composition of RARESYNC and the aforementioned view core. Since the view core requires 8 communication steps in order for correct processes to decide, a synchronous overlap of 8δ is sufficient. Thus, we parameterize RARESYNC with $\Delta = 8\delta$ (line 3). In short, the view core is subservient to RARESYNC, i.e., when RARESYNC triggers the advance(v) event (line 7), the view core starts executing the logic of view v (line 8). Once the view core decides (line 9), QUAD decides (line 10).

5.1.3 Proof overview

The agreement and weak validity properties of QUAD are ensured by the view core's implementation. As for the termination property, the view core, and therefore QUAD, is guaranteed to decide as soon as processes have synchronized in the same view with a correct leader for $\Delta = 8\delta$ time at or after GST. Since RARESYNC ensures the eventual synchro-

Algorithm 3 QUAD: Pseudocode (for process P_i)

```
1: Modules:
2:    View_Core core
3:    View_Synchronizer sync ← RARESYNC($\Delta = 8\delta$)
4: upon init(Value proposal):                              ▷ propose proposal
5:    core.init(proposal)                                  ▷ initialize the view core
6:    sync.init                                            ▷ start RARESYNC
7: upon synchronizer.advance(View $v$):
8:    core.start_executing($v$)
9: upon core.decide(Value decision):
10:   trigger decide(decision)                             ▷ decide decision
```

nization property, this eventually happens, which implies that QUAD satisfies termination. As processes synchronize within $O(f \cdot \delta)$ time after GST, the latency complexity of QUAD is $O(f \cdot \delta)$.

As for the total communication complexity, it is the sum of the communication complexity of (1) RARESYNC, which is $O(n^2)$, and (2) the view core, which is also $O(n^2)$. The view core's complexity is a consequence of the fact that:

- Each process executes $O(1)$ epochs between GST and the time by which every process decides,
- Each epoch has $f + 1$ views,
- A process can be the leader in only one view of any epoch, and
- A process sends $O(n)$ words in a view if it is the leader, and $O(1)$ words otherwise, for an average of $O(1)$ words per view in any epoch.

Thus, the view core's communication complexity is $O(n^2) = O(1) \cdot (f + 1) \cdot O(1) \cdot n$. Therefore, QUAD indeed achieves $O(n^2)$ communication complexity. In summary:
Theorem: QUAD *is a Byzantine consensus protocol ensuring weak validity with (1) $O(n^2)$ communication complexity, and (2) $O(f \cdot \delta)$ latency complexity.*

5.2 QUAD: formal proof

In this section, we give the complete pseudocode of QUAD's view core module (algorithms 4 and 5), and we formally prove that QUAD solves consensus (with weak validity) with $O(n^2)$ communication complexity and $O(f \cdot \delta)$ latency complexity.

5.2.1 Proof of correctness

In this paragraph, we show that QUAD ensures weak validity, termination and agreement. Recall that the main body of QUAD is given in Algorithm 3, whereas its view synchronizer RARESYNC is presented in Algorithm 2 and its view core in Algorithm 5. We underline that the proofs concerned with the view core of QUAD can be found in [13], as QUAD uses the same view core as HotStuff.

Algorithm 4 QUAD: View core's utilities (for process P_i)

```
1: function msg(String type, Value value, Quorum_Certificate qc, View view):
2:    m.type ← type; m.value ← value; m.qc ← qc; m.view ← view
3:    return m
4: function vote_msg(String type, Value value, Quorum_Certificate qc, View view):
5:    m ← msg(type, value, qc, view)
6:    m.partial_sig ← ShareSign_i([m.type, m.value, m.view])
7:    return m
8: ▷ All the messages in M have the same type, value and view
9: function qc(Set(Vote_Message) M):
10:     qc.type ← m.type, where $m \in M$
11:     qc.value ← m.value, where $m \in M$
12:     qc.view ← m.view, where $m \in M$
13:     qc.sig ← Combine({partial_sig | partial_sig is in a message that belongs to M})
14:     return qc
15: function matching_msg(Message m, String type, View view):
16:     return m.type = type and m.view = view
17: function matching_qc(Quorum_Certificate qc, String type, View view):
18:     return qc.type = type and qc.view = view
```

We start by proving that QUAD ensures weak validity.

Theorem 3 *(Weak validity)* QUAD *ensures weak validity.*

Proof Suppose that all processes are correct. Whenever a correct process updates its *prepareQC* variable (line 25 of Algorithm 5), it updates it to a quorum certificate vouching for a proposed value. Therefore, leaders always propose a proposed value since the proposed value is "formed" out of *prepareQC*s of processes (line 9 of Algorithm 5). Given that a correct process executes line 43 of Algorithm 5 for a value proposed by the leader of the current view, which is proposed by a process (recall that all processes are correct), the weak validity property is ensured. □

Next, we prove agreement.

Theorem 4 *(Agreement)* QUAD *ensures agreement.*

Proof Two conflicting quorum certificates associated with the same view cannot be obtained in the view core of QUAD (Algorithm 5); otherwise, a correct process would vote for both certificates, which is not possible according to Algorithm 5. Therefore, two correct processes cannot decide different values from the view core of QUAD in the same view. Hence, we need to show that, if a correct process decides v in some view *view* in the view core (line 43 of Algorithm 5), then no conflicting quorum certificate can be obtained in the future views.

Since a correct process decides v in view *view* in the view core, the following holds at $f + 1$ correct processes: $lockedQC.value = v$ and $lockedQC.view = view$ (line 34 of Algorithm 5). In order for another correct process to decide a different value in some future view, a prepare quorum certificate for a value different than v must be obtained in a view greater than *view*. However, this is impossible as $f + 1$ correct processes whose $lockedQC.value = v$ and $lockedQC.view = view$ will not support such a prepare quorum certificate (i.e., the check at line 16 of Algorithm 5 will

Algorithm 5 QUAD: View core (for process P_i)

```
1: upon init(Value $proposal$):
2:     $proposal_i \leftarrow proposal$                                    ▷ $P_i$'s proposal
3: upon start_executing(View $view$):
4:     ▷ Prepare phase
5:     send msg(VIEW- CHANGE, $\bot$, $prepareQC$, $view$) to leader($view$)
6:     as leader($view$):
7:         wait for $2f+1$ VIEW- CHANGE messages:
8:             $M \leftarrow \{m \mid$ matching_msg($m$, VIEW- CHANGE, $view$)$\}$
9:         Quorum_Certificate $highQC \leftarrow qc$ with the highest $qc.view$ in $M$
10:        Value $proposal \leftarrow highQC.value$
11:        if $proposal = \bot$:
12:            $proposal \leftarrow proposal_i$          ▷ $proposal_i$ denotes the proposal of $P_i$
13:        broadcast msg(PREPARE, $proposal$, $highQC$, $view$)
14:    as a process:                 ▷ every process executes this part of the pseudocode
15:        wait for message $m$: matching_msg($m$, PREPARE, $view$) from leader($view$)
16:        if $m.qc.value = m.value$ and ($lockedQC.value = m.value$ or $m.qc.view > lockedQC.view$):
17:            send vote_msg(PREPARE, $m.value$, $\bot$, $view$) to leader($view$)
18:    ▷ Precommit phase
19:    as leader($view$):
20:        wait for $2f+1$ votes: $V \leftarrow \{vote \mid$ matching_msg($vote$, PREPARE, $view$)$\}$
21:        Quorum_Certificate $qc \leftarrow$ qc($V$)
22:        broadcast msg(PRECOMMIT, $\bot$, $qc$, $view$)
23:    as a process:                 ▷ every process executes this part of the pseudocode
24:        wait for message $m$: matching_qc($m.qc$, PREPARE, $view$) from leader($view$)
25:        $prepareQC \leftarrow m.qc$
26:        send vote_msg(PRECOMMIT, $m.qc.value$, $\bot$, $view$) to leader($view$)
27:    ▷ Commit phase
28:    as leader($view$):
29:        wait for $2f+1$ votes: $V \leftarrow \{vote \mid$ matching_msg($vote$, PRECOMMIT, $view$)$\}$
30:        Quorum_Certificate $qc \leftarrow$ qc($V$)
31:        broadcast msg(COMMIT, $\bot$, $qc$, $view$)
32:    as a process:                 ▷ every process executes this part of the pseudocode
33:        wait for message $m$: matching_qc($m.qc$, PRECOMMIT, $view$) from leader($view$)
34:        $lockedQC \leftarrow m.qc$
35:        send vote_msg(COMMIT, $m.qc.value$, $\bot$, $view$) to leader($view$)
36:    ▷ Decide phase
37:    as leader($view$):
38:        wait for $2f+1$ votes: $V \leftarrow \{vote \mid$ matching_msg($vote$, COMMIT, $view$)$\}$
39:        Quorum_Certificate $qc \leftarrow$ qc($V$)
40:        broadcast msg(DECIDE, $\bot$, $qc$, $view$)
41:    as a process:                 ▷ every process executes this part of the pseudocode
42:        wait for message $m$: matching_qc($m.qc$, COMMIT, $view$) from leader($view$)
43:        trigger decide($m.qc.value$)
```

return false). Thus, it is impossible for correct processes to disagree in the view core even across multiple views. The agreement property is ensured by QUAD. □

Finally, we prove termination.

Theorem 5 *(Termination)* QUAD *ensures termination.*

Proof RARESYNC ensures that, eventually, all correct processes remain in the same view *view* with a correct leader for (at least) $\Delta = 8\delta$ time after GST. When this happens, all correct processes decide in the view core.

Indeed, the leader of *view* learns the highest obtained locked quorum certificate through the VIEW- CHANGE messages (line 9 of Algorithm 5). Therefore, every correct process supports the proposal of the leader (line 17 of Algorithm 5) as the check at line 16 of Algorithm 5 returns true. After the leader obtains a prepare quorum certificate in *view*, all correct processes vote in the following phases of the same view. Thus, all correct processes decide from the view core (line 43 of Algorithm 5), which concludes the proof. □

Thus, QUAD indeed solves the Byzantine consensus problem with weak validity.

Corollary 1 QUAD *is a partially synchronous Byzantine consensus protocol ensuring weak validity.*

5.2.2 Proof of complexity

Next, we show that QUAD achieves $O(n^2)$ communication complexity and $O(f \cdot \delta)$ latency complexity. Before we start the proof, we clarify one point about Algorithm 3: as soon as advance(v) is triggered (line 7), for some view v, the process immediately stops accepting and sending messages for the previous view. In other words, it is as if the "stop accepting and sending messages for the previous view" action immediately follows the advance($\cdot$) upcall in Algorithm 2.[7]

We begin by proving that, if a correct process sends a message of the view core associated with a view v which belongs to an epoch e, then the last entered epoch prior to sending the message (in the behavior of the process) is e (this result is similar to the one of Lemma 16). A message is a *view-core* message if it is of the VIEW- CHANGE, PREPARE, PRECOMMIT, COMMIT or DECIDE type.

Lemma 26 *Let P_i be a correct process and let P_i send a view-core message associated with a view v, where v belongs to an epoch e. Then, e is the last epoch entered by P_i in β_i before sending the message.*

Proof Process P_i enters the view v before sending the view-core message (since start_executing(v) is invoked upon P_i entering v; line 8 of Algorithm 3). By Lemma 8, P_i enters the first view of the epoch e (and, hence, e) before sending the message. By contradiction, suppose that P_i enters another epoch e' after entering e and before sending the view-core message.

By Lemma 3, we have that $e' > e$. However, this means that P_i does not send any view-core messages associated with v after entering e' (since $(e'-1)\cdot(f+1)+1 > v$ and P_i enters monotonically increasing views by Lemma 3). Thus, a contradiction, which concludes the proof. □

Next, we show that a correct process sends (at most) $O(n)$ view-core messages associated with a single epoch.

Lemma 27 *Let P_i be a correct process. For any epoch e, P_i sends (at most) $O(n)$ view-core messages associated with views that belong to e.*

[7] Note that this additional action does not disrupt RARESYNC (nor its proof of correctness and complexity).

Proof Recall that P_i enters monotonically increasing views (by Lemma 3), which means that P_i never invokes start_executing(v) (line 8 of Algorithm 3) multiple times for any view v.

Consider a view v that belongs to e. We consider two cases:

- Let P_i be the leader of v. In this case, P_i sends (at most) $O(n)$ view-core messages associated with v.
- Let P_i not be the leader of v. In this case, P_i sends (at most) $O(1)$ view-core messages associated with v.

Given that P_i is the leader of at most one view in every epoch e (since leader($\cdot$) is a round-robin function), P_i sends (at most) $1 \cdot O(n) + f \cdot O(1) = O(n)$ view-core messages associated with views that belong to e. □

Finally, we prove the complexity of QUAD.

Theorem 6 *(Complexity)* QUAD *achieves $O(n^2)$ communication complexity and $O(f \cdot \delta)$ latency complexity.*

Proof As soon as all correct processes remain in the same view for 8δ time, all correct processes decide from the view core. As RARESYNC uses $\Delta = 8\delta$ in the implementation of QUAD (line 3 of Algorithm 3), all processes decide by time $t_s + 8\delta$, where t_s is the first synchronization time after *GST* (Definition 3). Given that $t_s + 8\delta - GST$ is the latency of RARESYNC (see Sect. 4.1) and the latency complexity of RARESYNC is $O(f \cdot \delta)$ (by Theorem 2), the latency complexity of QUAD is indeed $O(f \cdot \delta)$.

Fix a correct process P_i. For every epoch e, P_i sends (at most) $O(n)$ view-core messages associated with views that belong to e (by Lemma 27). Moreover, if P_i sends a view-core message associated with a view that belongs to an epoch e, then e is the last epoch entered by P_i prior to sending the message (by Lemma 26). Hence, in the time period $[GST, t_s + 8\delta]$, P_i sends view-core messages associated with views that belong to (at most) $O(1)$ epochs (by Lemma 25). Thus, P_i sends (at most) $O(1) \cdot O(n) = O(n)$ view-core messages in the time period $[GST, t_s + 8\delta]$, each containing a single word. Moreover, during this time period, the communication complexity of RARESYNC is $O(n^2)$ (by Theorem 2). Therefore, the communication complexity of QUAD is $n \cdot O(n) + O(n^2) = O(n^2)$. □

As a final note, while our definition of communication complexity considers the words exchanged between *GST* until $t_d(\alpha)$ (i.e., when the last correct process has decided), it is straightforward to extend our results to account for all messages exchanged between *GST* and infinity. This is achieved by having correct processes halt the sending of messages of the underlying RARESYNC protocol immediately after deciding, and rebroadcasting (once) the first correct DECIDE message they see. This is because, by the time the first correct process decides, every correct process is also guaranteed to decide regardless of the continued use of RARESYNC, as termination is now solely dependent on the reception of the DECIDE message. This once-per-process rebroadcast incurs only an additional $O(n^2)$ words exchanged in total, with latency remaining unaffected, so all results continue to hold.

5.3 SQUAD: protocol description

At last, we present SQUAD, which we derive from QUAD.

5.3.1 Deriving SQUAD from QUAD

Imagine a locally-verifiable, *constant-sized* cryptographic proof σ_v vouching that value v is *valid*. Moreover, imagine that it is impossible, in the case in which all correct processes propose v to QUAD, for any process to obtain a proof for a value different from v:

- Computability: If all correct processes propose v to QUAD, then no process (even if faulty) obtains a cryptographic proof $\sigma_{v'}$ for a value $v' \neq v$.

If such a cryptographic primitive were to exist, then the QUAD protocol could be modified in the following manner in order to satisfy the validity property introduced in Sect. 1:

- A correct process accompanies each value by a cryptographic proof that the value is valid.
- A correct process ignores any message with a value not accompanied by the value's proof.

Suppose that all correct processes propose the same value v and that a correct process P_i decides v' from the modified version of QUAD. Given that P_i ignores messages with non-valid values, P_i has obtained a proof for v' before deciding. The computability property of the cryptographic primitive guarantees that $v' = v$, implying that validity is satisfied. Given that the proof is of constant size, the communication complexity of the modified version of QUAD remains $O(n^2)$.

Therefore, the main challenge in obtaining SQUAD from QUAD, while preserving QUAD's complexity, lies in implementing the introduced cryptographic primitive.

5.3.2 Certification phase

SQUAD utilizes its *certification phase* (Algorithm 6) to obtain the introduced constant-sized cryptographic proofs; we call these proofs *certificates*.[8] Formally, Certificate denotes the

[8] Note the distinction between certificates and prepare and locked QCs of the view core.

set of all certificates. Moreover, we define a locally computable function $\mathsf{verify}\colon \mathsf{Value} \times \mathsf{Certificate} \to \{\mathit{true}, \mathit{false}\}$. We require the following properties to hold:

- *Computability:* If all correct processes propose the same value v to SQUAD, then no process (even if faulty) obtains a certificate $\sigma_{v'}$ with $\mathsf{verify}(v', \sigma_{v'}) = \mathit{true}$ and $v' \neq v$.
- *Liveness:* Every correct process eventually obtains a certificate σ_v such that $\mathsf{verify}(v, \sigma_v) = \mathit{true}$, for some value v.

The computability property states that, if all correct processes propose the same value v to SQUAD, then no process (even if Byzantine) can obtain a certificate for a value different from v. The liveness property ensures that all correct processes eventually obtain a certificate. Hence, if all correct processes propose the same value v, all correct processes eventually obtain a certificate for v and no process obtains a certificate for a different value.

In order to implement the certification phase, we assume an $(f+1, n)$-threshold signature scheme (see Sect. 3) used throughout the entirety of the certification phase. The $(f+1, n)$-threshold signature scheme allows certificates to count as a single word, as each certificate is a threshold signature. Finally, in order to not disrupt QUAD's communication and latency, the certification phase itself incurs $O(n^2)$ communication and $O(1)$ latency.

Algorithm 6 Certification Phase: Pseudocode (for process P_i)

```
1:  upon init(Value $proposal$):                               ▷ propose value $proposal$
2:      ▷ inform other processes that $proposal$ was proposed
3:      broadcast ⟨DISCLOSE, $proposal$, $ShareSign_i(proposal)$⟩
4:  upon exists Value $v$ such that ⟨DISCLOSE, $v$, P_Signature $sig$⟩ is received from $f+1$
    processes:
5:      ▷ a certificate for $v$ is obtained
6:      Certificate $\sigma_v \leftarrow Combine(\{sig \mid sig$ is received in a DISCLOSE message$\})$
7:      broadcast ⟨CERTIFICATE, $v$, $\sigma_v$⟩                   ▷ disseminate the certificate
8:      exit the certification phase
9:  upon for the first time (1) DISCLOSE message is received from $2f+1$ processes,
    and (2) not exist Value $v$ such that ⟨DISCLOSE, $v$, P_Signature $sig$⟩ is received from
    $f+1$ processes:
10:     ▷ inform other processes that any value can be "accepted"
11:     broadcast ⟨ALLOW- ANY, $ShareSign_i$("any value")⟩
12: upon ⟨ALLOW- ANY, P_Signature $sig$⟩ is received from $f+1$ processes :
13:     ▷ a certificate for "any value" is obtained
14:     Certificate $\sigma_\perp \leftarrow Combine(\{sig \mid$
    $sig$ is received in an ALLOW- ANY message$\})$
15:     broadcast ⟨CERTIFICATE, $\perp$, $\sigma_\perp$⟩                 ▷ disseminate the certificate
16:     exit the certification phase
17: ▷ a certificate for $v$ is obtained; $v$ can be $\perp$, meaning that $\sigma_v$ vouches for any value
18: upon reception of ⟨CERTIFICATE, Value $v$, Certificate $\sigma_v$⟩:
19:     broadcast ⟨CERTIFICATE, $v$, $\sigma_v$⟩                   ▷ disseminate the certificate
20:     exit the certification phase
21: function verify(Value $v$, Certificate $\sigma$):
22:     if $CombinedVerify$("any value", $\sigma$) = $true$: return $true$
23:     else if $CombinedVerify(v, \sigma) = true$: return $true$
24:     else return $false$
```

A certificate σ vouches for a value v (the $\mathsf{verify}(\cdot)$ function at line 21) if (1) σ is a threshold signature of the predefined string "any value" (line 22), or (2) σ is a threshold signature of v (line 23). Otherwise, $\mathsf{verify}(v, \sigma)$ returns *false*.

Once P_i enters the certification phase (line 1), P_i informs all processes about the value it has proposed by broadcasting a DISCLOSE message (line 3). Process P_i includes a partial signature of its proposed value in the message. If P_i receives DISCLOSE messages for the same value v from $f+1$ processes (line 4), P_i combines the received partial signatures into a threshold signature of v (line 6), which represents a certificate for v. To ensure liveness, P_i disseminates the certificate (line 7).

If P_i receives $2f+1$ DISCLOSE messages and there does not exist a "common" value received in $f+1$ (or more) DISCLOSE messages (line 9), the process concludes that it is fine for a certificate for *any* value to be obtained. Therefore, P_i broadcasts an ALLOW- ANY message containing a partial signature of the predefined string "any value" (line 11).

If P_i receives $f+1$ ALLOW- ANY messages (line 12), it combines the received partial signatures into a certificate that vouches for *any* value (line 14), and it disseminates the certificate (line 15). Since ALLOW- ANY messages are received from $f+1$ processes, there exists a correct process that has verified that it is indeed fine for such a certificate to exist.

If, at any point, P_i receives a certificate (line 18), it adopts the certificate, and disseminates it (line 19) to ensure liveness.

Given that each message of the certification phase contains a single word, the certification phase incurs $O(n^2)$ communication. Moreover, each correct process obtains a certificate after (at most) $2 = O(1)$ rounds of communication. Therefore, the certification phase incurs $O(1)$ latency.

We explain below why the certification phase (Algorithm 6) ensures computability and liveness:

- Computability: If all correct processes propose the same value v to SQUAD, all correct processes broadcast a DISCLOSE message for v (line 3). Since $2f+1$ processes are correct, no process obtains a certificate $\sigma_{v'}$ for a value $v' \neq v$ such that $CombinedVerify(v', \sigma_{v'}) = true$ (line 23). Moreover, as every correct process receives $f+1$ DISCLOSE messages for v within any set of $2f+1$ received DISCLOSE messages, no correct process sends an ALLOW- ANY message (line 11). Hence, no process obtains a certificate $\sigma_\perp$ such that $CombinedVerify(\text{"any value"}, \sigma_\perp) = true$ (line 22). Thus, computability is ensured.
- Liveness: If a correct process receives $f+1$ DISCLOSE messages for a value v (line 4), the process obtains a certificate for v (line 6). Since the process disseminates the certificate (line 7), every correct process eventually obtains a certificate (line 18), ensuring liveness in this scenario. Otherwise, all correct processes broadcast an

ALLOW- ANY message (line 11). Since there are at least $2f+1$ correct processes, every correct process eventually receives $f+1$ ALLOW- ANY messages (line 12), thus obtaining a certificate. Hence, liveness is satisfied in this case as well.

5.3.3 SQUAD = Certification phase + QUAD

We obtain SQUAD by combining the certification phase with QUAD. The pseudocode of SQUAD is given in Algorithm 7.

Algorithm 7 SQUAD: Pseudocode (for process P_i)

```
1: upon init(Value proposal):                                    ▷ propose value proposal
2:     start the certification phase with proposal
3: upon exiting the certification phase with a certificate $\sigma_v$ for a value $v$:
4:     ▷ in QUAD$_{cer}$, processes ignore messages with values not accompanied by their certificates
5:     start executing QUAD$_{cer}$ with the proposal $(v, \sigma_v)$
6: upon QUAD$_{cer}$ decides Value decision:
7:     trigger decide(decision)                                  ▷ decide value decision
```

A correct process P_i executes the following steps in SQUAD:

1. P_i starts executing the certification phase with its proposal (line 2).
2. Once the process exits the certification phase with a certificate σ_v for a value v, it proposes (v, σ_v) to QUAD$_{cer}$, a version of QUAD "enriched" with certificates (line 5). While executing QUAD$_{cer}$, correct processes *ignore* messages containing values not accompanied by their certificates.
3. Once P_i decides from QUAD$_{cer}$ (line 6), P_i decides the same value from SQUAD (line 7).

In summary:
Theorem: SQUAD *is a Byzantine consensus protocol with (1) $O(n^2)$ communication complexity, and (2) $O(f \cdot \delta)$ latency complexity.*

5.4 SQUAD: formal proof

First, we show that the certification phase of SQUAD ensures computability and liveness.

Lemma 28 *(Computability & liveness) Certification phase (Algorithm 6) ensures computability and liveness. Moreover, every correct process sends (at most) $O(n)$ words and obtains a certificate by time $GST + 2\delta$.*

Proof As every correct process broadcasts DISCLOSE, CERTIFICATE or ALLOW- ANY messages at most once and each message contains a single word, every correct process sends (at most) $3 \cdot n \cdot 1 = O(n)$ words. Next, we prove computability and liveness.

Computability. Let all correct processes propose the same value v to SQUAD. Since no correct process broadcasts a DISCLOSE message for a value $v' \neq v$, no process ever obtains a certificate $\sigma_{v'}$ for v' such that $CombinedVerify(v', \sigma_{v'}) = true$ (line 23).

Since all correct processes broadcast a DISCLOSE message for v (line 3), the rule at line 9 never activates at a correct process. Thus, no correct process ever broadcasts an ALLOW- ANY message (line 11), which implies that no process obtains a certificate $\sigma_\perp$ such that $CombinedVerify(\text{"allow any"}, \sigma_\perp) = true$ (line 22). The computability property is ensured.

Liveness. Every correct process receives all DISCLOSE messages sent by correct processes by time $GST + \delta$ (since message delays are δ after GST; see Sect. 3). Hence, all correct processes receive (at least) $2f+1$ DISCLOSE messages by time $GST + \delta$. Therefore, by time $GST + \delta$, all correct processes send either (1) a CERTIFICATE message upon receiving $f+1$ DISCLOSE messages for the same value (line 7), or (2) an ALLOW- ANY message upon receiving $2f+1$ DISCLOSE messages without a "common value" (line 11). Let us consider two possible scenarios:

- There exists a correct process that has broadcast a CERTIFICATE message upon receiving $f+1$ DISCLOSE messages for the same value (line 7) by time $GST + \delta$. Every correct process receives this message by time $GST + 2\delta$ (line 18) and obtains a certificate. Liveness is satisfied by time $GST + 2\delta$ in this case.
- Every correct process broadcasts an ALLOW- ANY message (line 11) by time $GST + \delta$. Hence, every correct process receives $f+1$ ALLOW- ANY messages by time $GST + 2\delta$ (line 12) and obtains a certificate (line 14). The liveness property is guaranteed by time $GST + 2\delta$ in this case as well.

The liveness property is ensured by time $GST + 2\delta$. □

Finally, we show that SQUAD is a Byzantine consensus protocol with $O(n^2)$ communication complexity and $O(f \cdot \delta)$ latency complexity.

Theorem 7 SQUAD *is a Byzantine consensus protocol with (1) $O(n^2)$ communication complexity, and (2) $O(f \cdot \delta)$ latency complexity.*

Proof If a correct process decides a value v' and all correct processes have proposed the same value v, then $v' = v$ since (1) correct processes ignore values not accompanied by their certificates (line 5), and (2) the certification phase of SQUAD ensures computability (by Lemma 28). Therefore, SQUAD ensures validity.

Fix an execution E_{SQUAD} of SQUAD. We denote by t_{last} the time the last correct process starts executing QUAD_{cer} (line 5) in E_{SQUAD}; i.e., by t_{last} every correct process has exited the certification phase. Moreover, we denote the global stabilization time of E_{SQUAD} by GST_1. Now, we consider two possible scenarios:

- Let $GST_1 \geq t_{last}$. QUAD_{cer} solves the Byzantine consensus problem with $O(n^2)$ communication and $O(f \cdot \delta)$ latency (by Theorem 6). As processes send (at most) $O(n)$ words associated with the certification phase (by Lemma 28), consensus is solved in E_{SQUAD} with $n \cdot O(n) + O(n^2) = O(n^2)$ communication complexity and $O(f \cdot \delta)$ latency complexity.
- Let $GST_1 < t_{last}$. Importantly, $t_{last} - GST_1 \leq 2\delta$ (by Lemma 28). Now, we create an execution E_{QUAD} of the original QUAD protocol in the following manner:
  1. $E_{\text{QUAD}} \leftarrow E_{\text{SQUAD}}$. If a process sends a value with a valid accompanying certificate, then *just* the certificate is removed in E_{QUAD} (i.e., the corresponding message stays in E_{QUAD}). Otherwise, the entire message is removed. Note that no message sent by a correct process in E_{SQUAD} is removed from E_{QUAD} as correct processes only send values accompanied by their valid certificates.
  2. We remove from E_{QUAD} all events associated with the certification phase of SQUAD.
  3. The global stabilization time of E_{QUAD} is set to t_{last}. We denote this time by $GST_2 = t_{last}$. Note that we can set GST_2 to t_{last} as $t_{last} > GST_1$.

  In E_{QUAD}, consensus is solved with $O(n^2)$ communication and $O(f \cdot \delta)$ latency. Therefore, the consensus problem is solved in E_{SQUAD}. Let us now analyze the complexity of E_{SQUAD}:

  – The latency complexity of E_{SQUAD} is $t_{last} - GST_1 + O(f \cdot \delta) = O(f \cdot \delta)$ (as $t_{last} - GST_1 \leq 2\delta$).
  – The communication complexity of E_{SQUAD} is the sum of (1) the number of words sent in the time period $[GST_1, t_{last})$, and (2) the number of words sent at and after t_{last} and before the decision, which is $O(n^2)$ since that is the communication complexity of E_{QUAD} and each correct process sends (at most) $O(n)$ words associated with the certification phase (by Lemma 28).
    Fix a correct process P_i. Let us take a closer look at the time period $[GST_1, t_{last})$:
    - Let $epochs_{\text{RARESYNC}}$ denote the number of epochs for which P_i sends EPOCH- COMPLETED or ENTER- EPOCH messages in this time period. By Lemma 20, P_i enters (at most) $2 = O(1)$ epochs in this time period. Hence, $epochs_{\text{RARESYNC}} = O(1)$ (by Lemmas 16 and 17).
    - Let $epochs_{\text{QUAD}_{cer}}$ denote the number of epochs for which P_i sends view-core messages in this time period. By Lemma 20, P_i enters (at most) $2 = O(1)$ epochs in this time period. Hence, $epochs_{\text{QUAD}_{cer}} = O(1)$ (by Lemma 26).

    For every epoch e, P_i sends (at most) $O(n)$ EPOCH- COMPLETED and ENTER- EPOCH messages (by Lemmas 18 and 19). Moreover, for every epoch e, P_i sends (at most) $O(n)$ view-core messages associated with views that belong to e (by Lemma 27).[9] As each EPOCH- COMPLETED, ENTER- EPOCH and view-core message contains a single word and P_i sends at most $O(n)$ words during the certification phase (by Lemma 28), we have that P_i sends (at most) $epochs_{\text{RARESYNC}} \cdot O(n) + epochs_{\text{QUAD}_{cer}} \cdot O(n) + O(n) = O(n)$ words during the time period $[GST_1, t_{last})$. Therefore, the communication complexity of E_{SQUAD} is $n \cdot O(n) + O(n^2) + O(n^2) = O(n^2)$.[10]

  Hence, consensus is indeed solved in E_{SQUAD} with $O(n^2)$ communication complexity and $O(f \cdot \delta)$ latency complexity.

The theorem holds. □

6 Concluding remarks

This paper shows that the Dolev-Reischuk lower bound can be met by a partially synchronous Byzantine consensus protocol. Namely, we introduce SQUAD, an optimally-resilient partially synchronous Byzantine consensus protocol with optimal $O(n^2)$ communication complexity, and optimal $O(f \cdot \delta)$ latency complexity. SQUAD owes its complexity to RARESYNC, an "epoch-based" view synchronizer ensuring synchronization with quadratic communication and linear latency in partial synchrony. In the future, we aim to address the following limitations of RARESYNC.

[9] Note that Lemmas 16, 17, 18, 19, 20, 26 and 27, which we use to prove the theorem, assume that all correct processes have started executing RARESYNC and QUAD by GST. In Theorem 7, this might not be true as some processes might start executing RARESYNC after GST (since $t_{last} > GST$). However, it is not hard to verify that the claims of these lemmas hold even in this case.

[10] The first "$n \cdot O(n)$" term corresponds to the messages sent during the time period $[GST_1, t_{last})$, the second "$O(n^2)$" term corresponds to the messages sent during the certification phase, and the third "$O(n^2)$" term corresponds to the messages sent at and after t_{last} and before the decision has been made.

6.1 Lack of adaptiveness

RARESYNC is not *adaptive*, i.e., its complexity does not depend on the *actual* number b, but rather on the upper bound f, of Byzantine processes. Consider a scenario S in which all processes are correct; we separate them into three disjoint groups: (1) group A, with $|A| = f$, (2) group B, with $|B| = f$, and (3) group C, with $|C| = f + 1$. At GST, group A is in the first view of epoch e_{max}, group B is in the second view of e_{max}, and group C is in the third view of e_{max}.[11] Unfortunately, it is impossible for processes to synchronize in epoch e_{max}. Hence, they will need to wait for the end of epoch e_{max} in order to synchronize in the next epoch: thus, the latency complexity is $O(f \cdot \delta)$ (since e_{max} has $f + 1$ views) and the communication complexity is $O(n^2)$ (because of the "all-to-all" communication step at the end of e_{max}). In contrast, the view synchronizer presented in [49] achieves $O(1)$ latency and $O(n)$ communication complexity in S. Lastly, since our algorithm is not adaptive in terms of latency, i.e., its latency is $O(f \cdot \delta)$, where (crucially) f denotes the *upper bound* on the number of tolerated failures, and not the number of *actual* failures, our algorithm is also not *optimistically responsive* [13]: the latency of SQUAD depends on the maximum network delay δ and not on the actual speed of the network.

6.2 Suboptimal expected complexity

A second limitation of RARESYNC is that its *expected complexity* is the same as its worst-case complexity. Namely, the expected complexity considers a weaker adversary which does not have a knowledge of the $\mathsf{leader}(\cdot)$ function. Therefore, this adversary is unable to corrupt f processes that are scheduled to be leaders right after GST.

As the previously introduced scenario S does not include any Byzantine process, we can analyze it for the expected complexity of RARESYNC. Therefore, the expected latency complexity of RARESYNC is $O(f \cdot \delta)$ and the expected communication complexity of RARESYNC is $O(n^2)$. On the other hand, the view synchronizer of Naor and Keidar [49] achieves $O(1)$ expected latency complexity and $O(n)$ expected communication complexity.

6.3 Limited clock drift tolerance

A third limitation of RARESYNC is that its latency is susceptible to clock drifts. Namely, let $\phi > 1$ denote the bound on clock drifts after GST. To accommodate for the bounded clock drifts after GST, RARESYNC increases the duration of a view. The duration of the i-th view of an epoch becomes $\phi^i \cdot view\_duration$ (instead of only $view\_duration$). Thus, the latency complexity of RARESYNC becomes $O(f \cdot \delta \cdot \phi^f)$.

[11] Recall that e_{max} is the greatest epoch entered by a correct process before GST; see Sect. 4.3.

Acknowledgements The authors would like to thank Gregory Chockler and Alexey Gotsman for helpful conversations.

Funding Open access funding provided by EPFL Lausanne. This work was supported in part by the Hasler Foundation (#21084), by Singapore MOE grant MOE-T2EP20122-0014, and by the ARC Future Fellowship funding scheme (#180100496).

Data availability Data sharing is not applicable to this article as no datasets were generated or analysed in this work.

Declarations

Conflict of interest The authors have no conflicts of interest to declare that are relevant to the content of this article.

A Threshold signature scheme: formal definition

This section defines a (k, n)-threshold signature scheme ($k = 2f+1 = n-f$), which we (informally) introduced in Sect. 3. The threshold signature scheme and its properties are defined with respect to a security parameter $\kappa \in \mathbb{N}$. In the rest of the section, we say that a local protocol is *efficient* if and only if its complexity belongs to $poly(\kappa)$. Moreover, String denotes the set of all strings, $\mathcal{P}$ denotes the set of all partial signatures and $\mathcal{T}$ denotes the set of all threshold signatures.

Formally, a (k, n)-threshold signature scheme is a tuple consisting of:

1. $\mathsf{Keys} = \big(\mathsf{PK}, \mathsf{SK} = (sk_1, \dots, sk_n), \mathsf{VK} = (vk_1, \dots, vk_n)\big)$, where:
   - PK is a public key stored by every correct process.
   - SK is a vector of private key shares such that, for every correct process P_i, P_i stores its (and only its) private key share sk_i; sk_i is hidden from the adversary.
   - VK is a vector of verification keys and the entire vector is stored by every correct process.

2. $\mathit{ShareSign}(m \in \mathsf{String}, sk_i)$ is an efficient local (potentially probabilistic) protocol that takes (1) a string m, and (2) the private key share sk_i of a process P_i as the input. The protocol outputs a partial signature $\sigma_i^p \in \mathcal{P}$ of (at most) κ bits. In the main body of the paper, we used $\mathit{ShareSign}_i(m)$ to abbreviate $\mathit{ShareSign}(m, sk_i)$: $\mathit{ShareSign}_i(m) \equiv \mathit{ShareSign}(m, sk_i)$.
3. $\mathit{ShareVerify}(m \in \mathsf{String}, vk_i, \sigma_i^p \in \mathcal{P})$ is an efficient local deterministic protocol that takes (1) a string m, (2) the verification key vk_i of a process P_i, and (3) a partial signature σ_i^p as the input. The protocol outputs either *true* or *false* depending on whether σ_i^p is deemed as a valid partial signature of process P_i for m. In the main body of the paper, we used $\mathit{ShareVerify}_i(m, \sigma_i^p)$ to abbreviate $\mathit{ShareSign}(m, vk_i, \sigma_i^p)$: $\mathit{ShareVerify}_i(m, \sigma_i^p) \equiv \mathit{ShareVerify}(m, vk_i, \sigma_i^p)$.
4. $\mathit{Combine}(m \in \mathsf{String}, P \subset \mathcal{P})$, where P is a set of $|P| = k$ partial signatures of k distinct processes, is an efficient local protocol that takes (1) a string m, and (2) a set P of $|P| = k$ partial signatures as the input. The protocol outputs a threshold signature $\sigma^t \in \mathcal{T}$.
5. $\mathit{CombinedVerify}(m \in \mathsf{String}, \mathsf{PK}, \sigma^t \in \mathcal{T})$ is an efficient local deterministic protocol that takes (1) a string m, (2) the public key PK, and (3) a threshold signature σ^t as the input. The protocol outputs *true* or *false* depending on whether σ^t is deemed as a valid threshold signature for m. In the main body of the paper, we used $\mathit{CombinedVerify}(m, \sigma^t)$ to abbreviate $\mathit{CombinedVerify}(m, \mathsf{PK}, \sigma^t)$: $\mathit{CombinedVerify}(m, \sigma^t) \equiv \mathit{CombinedVerify}(m, \mathsf{PK}, \sigma^t)$.

The following guarantees are provided:

- *Correctness of partial signatures:* For every $i \in [1, n]$, $\mathit{ShareVerify}_i\big(m \in \mathsf{String}, \mathit{ShareSign}_i(m) \in \mathcal{P}\big)$ returns *true*.
- *Unforgeability of partial signatures:* If $\mathit{ShareVerify}_i(m \in \mathsf{String}, \sigma_i^p \in \mathcal{P})$ returns *true*, then (1) $\sigma_i^p \leftarrow \mathit{ShareSign}_i(m)$ has been executed by the process P_i, or (2) P_i is faulty.
- *Correctness of threshold signatures:* Let $m \in \mathsf{String}$ be any string and let S be any set of $|S| = k$ processes. Let $P = \{\sigma_i^p \,|\, \sigma_i^p \leftarrow \mathit{ShareSign}_i(m) \wedge P_i \in S\}$. Then, $\mathit{CombinedVerify}\big(m, \mathit{Combine}(m, P)\big)$ returns *true*.
- *Unforgeability of threshold signatures:* If $\mathit{Verify}(m \in \mathsf{String}, \sigma^t \in \mathcal{T})$ returns *true*, then there exists a set S of $|S| = k$ processes such that, for every process $P_i \in S$, (1) $\mathit{ShareSign}_i(m)$ has been executed by P_i, or (2) P_i is faulty.

B RareSync: basic properties

This section is dedicated to the correctness of RareSync (Algorithm 2). Concretely, we now formally prove Lemmas 1–8, which are merely stated in Sect. 4.4 without accompanying proofs for brevity.

Lemma 1 *Let P_i be a correct process. Then, $1 \leq view_i \leq f + 1$ throughout the entire execution.*

Proof First, $view_i \geq 1$ throughout the entire execution since (1) the initial value of $view_i$ is 1 (line 3 of Algorithm 1), and (2) the value of $view_i$ either increases (line 6) or is set to 1 (line 27).

By contradiction, suppose that $view_i = F > f + 1 > 1$ at some time during the execution. The update of $view_i$ to $F > f + 1$ must have been done at line 6. This means that, just before executing line 6, $view_i \geq f + 1$. However, this contradicts the check at line 5, which concludes the proof. □

Lemma 2 *Let P_i be a correct process. Let Exp_d be any expiration event of $dissemination\_timer_i$ that belongs to h_i and let Inv_d be the invocation of the* $\mathsf{measure}(\cdot)$ *method (on $dissemination\_timer_i$) that has produced Exp_d. Then, Exp_d immediately follows Inv_d in h_i.*

Proof In order to prove the lemma, we show that only Exp_d can immediately follow Inv_d in h_i. We consider the following scenarios:

- Let an invocation Inv_d' of the $\mathsf{measure}(\cdot)$ method on $dissemination\_timer_i$ immediately follow Inv_d in h_i: Inv_d' could only have been invoked either at line 18 or at line 24. However, an invocation of the $\mathsf{cancel}()$ method on $dissemination\_timer_i$ (line 17 or line 23) must immediately precede Inv_d' in h_i, which contradicts the fact that Inv_d immediately precedes Inv_d'. Therefore, this scenario is impossible.
- Let an invocation Inv_d' of the $\mathsf{cancel}()$ method on $dissemination\_timer_i$ immediately follow Inv_d in h_i: Inv_d' could only have been invoked either at line 17 or at line 23. However, an invocation of the $\mathsf{cancel}()$ method on $view\_timer_i$ (line 16 or line 22) must immediately precede Inv_d' in h_i, which contradicts the fact that Inv_d immediately precedes Inv_d'. Hence, this scenario is impossible, as well.
- Let an expiration event $Exp_d' \neq Exp_d$ of $dissemination\_timer_i$ immediately follow Inv_d in h_i: As Inv_d could have been invoked either at line 18 or at line 24, an invocation of the $\mathsf{cancel}()$ method on $dissemination\_timer_i$ (line 17 or line 23) immediately precedes Inv_d in h_i. This contradicts the fact that $Exp_d' \neq Exp_d$ is produced and immediately follows Inv_d, which renders this scenario impossible.
- Let an invocation Inv_v of the $\mathsf{measure}(\cdot)$ method on $view\_timer_i$ immediately follow Inv_d in h_i: Inv_v could have been invoked either at line 8 or at line 29. We further consider both cases:

  - If Inv_v was invoked at line 8, then Inv_v is immediately preceded by an expiration event of $view\_timer_i$ (line 4). This case is impossible as Inv_v is not immediately preceded by Inv_d.
  - If Inv_v was invoked at line 29, then Inv_v is immediately preceded by an expiration event of $dissemination\_timer_i$ (line 25). This case is also impossible as Inv_v is not immediately preceded by Inv_d.

  As neither of the two cases is possible, Inv_v cannot immediately follow Inv_d.
- Let an invocation Inv_v of the cancel() method on $view\_timer_i$ immediately follow Inv_d in h_i: Inv_v could have been invoked either at line 16 or at line 22. In both cases, an invocation of the cancel() method on $dissemination\_timer$ (line 17 or line 23) immediately follows Inv_v in h_i. This contradicts the fact that Inv_d produces Exp_d, which implies that this case is impossible.
- Let an expiration event Exp_v of $view\_timer_i$ immediately follow Inv_d in h_i: As Inv_d could have been invoked either at line 18 or at line 24, invocations of the cancel() method on $view\_timer_i$ and $dissemination\_timer_i$ (lines 16, 17 or lines 22, 23) immediately precede Inv_d in h_i. This contradicts the fact that Exp_v is produced and immediately follows Inv_d, which renders this scenario impossible.

As any other option is impossible, Exp_d must immediately follow Inv_d in h_i. Thus, the lemma. □

Lemma 3 *(Monotonically increasing views) Let P_i be a correct process. Let $e_1 =$ advance(v), $e_2 =$ advance(v') and $e_1 \overset{\beta_i}{\prec} e_2$. Then, $v' > v$.*

Proof Let $epoch_i = e$ and $view_i = j$ when P_i triggers advance(v). Moreover, let $epoch_i = e'$ and $view_i = j'$ when P_i triggers advance(v'). As the value of the $epoch_i$ variable only increases throughout the execution (lines 13, 15 and lines 19, 21), $e' \geq e$.

We investigate both possibilities:

- Let $e' > e$. In this case, the lemma follows from Lemma 1 and the fact that $(e'-1)\cdot(f+1)+j' > (e-1)\cdot(f+1)+j$, for every $j, j' \in [1, f+1]$.
- Let $e' = e$. Just before triggering advance(v) (line 3 or line 9 or line 30), P_i has invoked the measure$(\cdot)$ method on $view\_timer_i$ (line 2 or line 8 or line 29); we denote this invocation of the measure$(\cdot)$ method by Inv_v. Now, we investigate two possible scenarios:

  - Let P_i trigger advance(v') at line 9. By contradiction, suppose that $j' \leq j$. Hence, just before triggering advance(v') (i.e., just before executing line 6), we have that $view_i < j$. Thus, line 27 must have been executed by P_i after triggering advance(v) and before triggering advance(v'), which means that an expiration event of $dissemination\_timer_i$ (line 25) follows Inv_v in h_i. By Lemma 2, the measure$(\cdot)$ method on $dissemination\_timer_i$ was invoked by P_i after the invocation of Inv_v. Hence, when the aforementioned invocation of the measure$(\cdot)$ method on $dissemination\_timer_i$ was invoked by P_i (line 18 or line 24), the $epoch_i$ variable had a value greater than e (line 15 or line 21) since $epoch_i \geq e$ when processing line 13 or line 19; recall that the value of the $epoch_i$ variable only increases throughout the execution. Therefore, we reach a contradiction with the fact that $e' = e$, which means that $j' > j$ and the lemma holds in this case.
  - Let P_i trigger advance(v') at line 30. In this case, P_i processes an expiration event of $dissemination\_timer_i$ (line 25); therefore, the measure$(\cdot)$ method on $dissemination\_timer_i$ was invoked by P_i after the invocation of Inv_v (by Lemma 2). As in the previous case, when the aforementioned invocation of the measure$(\cdot)$ method on $dissemination\_timer_i$ was invoked by P_i (line 18 or line 24), the $epoch_i$ variable had a value greater than e (line 15 or line 21); recall that the value of the $epoch_i$ variable only increases throughout the execution. Thus, we reach a contradiction with the fact that $e' = e$, which renders this case impossible.

  In the only possible scenario, we have that $j' > j$, which implies that $v' > v$.

The lemma holds as it holds in both possible cases. □

Lemma 4 *Let P_i be a correct process. Let Inv_v be any invocation of the* measure$(\cdot)$ *method on $view\_timer_i$ that belongs to h_i. Invocation Inv_v is not immediately followed by another invocation of the* measure$(\cdot)$ *method on $view\_timer_i$ in $h_i\ |_{view}$.*

Proof We denote by Inv'_v the first invocation of the measure$(\cdot)$ method on $view\_timer_i$ after Inv_v in $h_i\ |_{view}$. If Inv'_v does not exist, the lemma trivially holds. Hence, let Inv'_v exist in the rest of the proof. We examine two possible cases:

- Let Inv'_v be invoked at line 8: In this case, there exists an expiration event of $view\_timer_i$ (line 4) separating Inv_v and Inv'_v in $h_i\ |_{view}$.
- Let Inv'_v be invoked at line 29: In this case, Inv'_v is immediately preceded by an expiration event Exp_d of $dissemination\_timer_i$ (line 25) in h_i. By Lemma 2, an invocation Inv_d of the measure$(\cdot)$ method on $dissemination\_timer_i$ immediately precedes Exp_d in h_i. As Inv_d could have been invoked either at line 18

or at line 24, Inv_d is immediately preceded by invocations of the cancel() methods on $view\_timer_i$ and $dissemination\_timer_i$ (lines 16, 17 or lines 22, 23). Hence, in this case, an invocation of the cancel() method on $view\_timer_i$ separates Inv_v and Inv'_v in $h_i\ |_{view}$.

The lemma holds since Inv'_v does not immediately follow Inv_v in $h_i\ |_{view}$ in any of the two cases. □

A direct consequence of Lemma 4 is that an expiration event of $view\_timer_i$ immediately follows (in a timer history associated with $view\_timer_i$) the measure(·) invocation that has produced it.

Lemma 5 *Let P_i be a correct process. Let Exp_v be any expiration event that belongs to $h_i\ |_{view}$ and let Inv_v be the invocation of the* measure(·) *method (on $view\_timer_i$) that has produced Exp_v. Then, Exp_v immediately follows Inv_v in $h_i\ |_{view}$.*

Proof We prove the lemma by induction.
Base step: *Let Inv_v^1 be the first invocation of the* measure(·) *method in $h_i\ |_{view}$ that produces an expiration event, and let Exp_v^1 be the expiration event produced by Inv_v^1. Expiration event Exp_v^1 immediately follows Inv_v^1 in $h_i\ |_{view}$.*
Since Inv_v^1 produces the expiration event Exp_v^1, an invocation of the cancel() method does not immediately follow Inv_v^1 in $h_i\ |_{view}$. Moreover, no invocation of the measure(·) method immediately follows Inv_v^1 in $h_i\ |_{view}$ (by Lemma 4). Finally, no expiration event produced by a different invocation of the measure(·) method immediately follows Inv_v^1 in $h_i\ |_{view}$ since Inv_v^1 is the first invocation of the method in $h_i\ |_{view}$ that produces an expiration event. Therefore, the statement of the lemma holds for Inv_v^1 and Exp_v^1.
Induction step: *Let Inv_v^j be the j-th invocation of the* measure(·) *method in $h_i\ |_{view}$ that produces an expiration event, where $j > 1$, and let Exp_v^j be the expiration event produced by Inv_v^j. Expiration event Exp_v^j immediately follows Inv_v^j in $h_i\ |_{view}$.*
Induction hypothesis: For every $k \in [1, j-1]$, the k-th invocation of the measure(·) *method in $h_i\ |_{view}$ that produces an expiration event is immediately followed by the produced expiration event in $h_i\ |_{view}$.*
An invocation of the cancel() method does not immediately follow Inv_v^j in $h_i\ |_{view}$ since Inv_v^j produces Exp_v^j. Moreover, no invocation of the measure(·) method immediately follows Inv_v^j in $h_i\ |_{view}$ (by Lemma 4). Lastly, no expiration event produced by a different invocation of the measure(·) method immediately follows Inv_v^j in $h_i\ |_{view}$ by the induction hypothesis. Therefore, the statement of the lemma holds for Inv_v^j and Exp_v^j, which concludes the proof. □

Lemma 6 *Let P_i be a correct process. Let Exp_v be any expiration event of $view\_timer_i$ that belongs to h_i and let Inv_v be the invocation of the* measure(·) *method (on $view\_timer_i$) that has produced Exp_v. Then, Exp_v immediately follows Inv_v in h_i.*

Proof Let us consider all possible scenarios (as in the proof of Lemma 2):

- Let an invocation Inv_d of the measure(·) method on $dissemination\_timer_i$ immediately follow Inv_v in h_i: Inv_d could have been invoked either at line 18 or at line 24. However, an invocation of the cancel() method on $dissemination\_timer_i$ (line 17 or line 23) must immediately precede Inv_d in h_i, which contradicts the fact that Inv_v immediately precedes Inv_d. Therefore, this scenario is impossible.
- Let an invocation Inv_d of the cancel() method on $dissemination\_timer_i$ immediately follow Inv_v in h_i: Inv_d could have been invoked either at line 17 or at line 23. However, an invocation of the cancel() method on $view\_timer_i$ (line 16 or line 22) must immediately precede Inv_d in h_i, which contradicts the fact that Inv_v immediately precedes Inv_d. Hence, this scenario is impossible, as well.
- Let an expiration event Exp_d of $dissemination\_timer_i$ immediately follow Inv_v in h_i: This is impossible due to Lemma 2.
- Let the event immediately following Inv_v be (1) an invocation of the measure(·) method on $view\_timer_i$, or (2) an invocation of the cancel() method on $view\_timer_i$, or (3) an expiration event Exp'_v of $view\_timer_i$, where $Exp'_v \neq Exp_v$: This case is impossible due to Lemma 5.

As any other option is impossible, Exp_v must immediately follow Inv_v in h_i. □

Lemma 7 *Let P_i be a correct process. Let Inv_v denote an invocation of the* measure(·) *method on $view\_timer_i$ which produces an expiration event, and let Exp_v denote the expiration event produced by Inv_v. Let $epoch_i = e$ and $view_i = v$ when P_i invokes Inv_v. Then, when P_i processes Exp_v (line 4), $epoch_i = e$ and $view_i = v$.*

Proof By contradiction, suppose that $epoch_i \neq e$ or $view_i \neq v$ when P_i processes Exp_v. Hence, the value of the variables of P_i must have changed between invoking Inv_v and processing Exp_v. Let us investigate all possible lines of Algorithm 2 where P_i could have modified its variables for the first time after invoking Inv_v (the first modification occurs before processing Exp_v):

- The $view_i$ variable at line 6: If P_i has modified its $view_i$ variable here, there exists an expiration event of $view\_timer_i$ (line 4) which follows Inv_v in h_i. By Lemma

6, this expiration event cannot occur before processing Exp_v, which implies that this case is impossible.
- The $epoch_i$ variable at line 15: If P_i updates its $epoch_i$ variable here, an invocation of the cancel() method on $view\_timer_i$ (line 16) separates Inv_v and Exp_v in h_i. However, this is impossible due to Lemma 6, which renders this case impossible.
- The $epoch_i$ variable at line 21: If P_i updates its $epoch_i$ variable here, an invocation of the cancel() method on $view\_timer_i$ (line 22) separates Inv_v and Exp_v in h_i. However, this is impossible due to Lemma 6, which implies that this case is impossible.
- The $view_i$ variable at line 27: If P_i updates its $view_i$ variable here, an expiration event of $dissemination\_timer_i$ (line 25) separates Inv_v and Exp_v in h_i, which contradicts Lemma 6.

Given that P_i does not change the value of neither $epoch_i$ nor $view_i$ between invoking Inv_v and processing Exp_v, the lemma holds. □

Lemma 8 *Let P_i be a correct process. Let* advance$(v) \in \beta_i$, *where v is the j-th view of an epoch e and $j > 1$. Then,* advance$(v-1) \overset{\beta_i}{\prec}$ advance(v).

Proof Since P_i enters view v, which is not the first view of epoch e, P_i triggers advance(v) at line 9: P_i could not have triggered advance(v) neither at line 3 nor at line 30 since v is not the first view of epoch e. Due to line 4, the measure$(\cdot)$ method was invoked on $view\_timer_i$ before advance(v) is triggered; we denote by Inv_v this specific invocation of the measure$(\cdot)$ method on $view\_timer_i$ and by Exp_v its expiration event (processed by P_i just before triggering advance(v)).

When P_i triggers advance(v) (at line 9), we have that $epoch_i = e$ and $view_i = j$. Moreover, when processing Exp_v, we have that $epoch_i = e$ and $view_i = j-1$ (due to line 6). By Lemma 7, when P_i has invoked Inv_v, we had the same state: $epoch_i = e$ and $view_i = j-1$. Process P_i could have invoked Inv_v either (1) at line 2, or (2) at line 8, or (3) at line 29. Since P_i triggers advance$(\cdot)$ immediately after (line 3, line 9, or line 30), that advance$(\cdot)$ indication is for $v-1$ (as $epoch_i = e$ and $view_i = j-1$ at that time). Hence, advance$(v-1) \overset{\beta_i}{\prec}$ advance(v). □

References

1. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982)
2. Crain, T., Gramoli, V., Larrea, M., Raynal, M.: DBFT: Efficient Byzantine consensus with a weak coordinator and its application to consortium blockchains. In: 17th IEEE International Symposium on Network Computing and Applications NCA, pp. 1–41 (2017)
3. Abraham, I., Malkhi, D., Nayak, K., Ren, L., Spiegelman, A.: Solida: a blockchain protocol based on reconfigurable byzantine consensus. In: Aspnes, J., Bessani, A., Felber, P., Leitão, J. (eds.) 21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, 18–20 Dec, 2017. LIPIcs, vol. 95, pp. 25–12519. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.OPODIS.2017.25
4. Gramoli, V.: From blockchain consensus back to Byzantine consensus. Future Gener. Comput. Syst. **107**, 760–769 (2020)
5. Lim, J., Suh, T., Gil, J., Yu, H.: Scalable and leaderless Byzantine consensus in cloud computing environments. Inf. Syst. Front. **16**(1), 19–34 (2014)
6. Dolev, D., Reischuk, R.: Bounds on information exchange for Byzantine agreement. J. ACM (JACM) **32**(1), 191–204 (1985)
7. Berman, P., Garay, J.A., Perry, K.J.: Bit optimal distributed consensus. In: Computer Science: Research and Applications, pp. 313–321. Springer, Boston (1992)
8. Momose, A., Ren, L.: Optimal communication complexity of authenticated Byzantine agreement. In: Gilbert, S. (ed.) 35th International Symposium on Distributed Computing, DISC 2021, 4–8 Oct, 2021, Freiburg, Germany (Virtual Conference). LIPIcs, vol. 209, pp. 32–13216. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.DISC.2021.32
9. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM (JACM) **35**(2), 288–323 (1988)
10. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. Assoc. Comput. Mach. **32**(2), 374–382 (1985)
11. Spiegelman, A.: In search for an optimal authenticated Byzantine agreement. In: Gilbert, S. (ed.) 35th International Symposium on Distributed Computing (DISC 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 209, pp. 38–13819. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.DISC.2021.38. https://drops.dagstuhl.de/opus/volltexte/2021/14840
12. Cohen, S., Keidar, I., Naor, O.: Byzantine agreement with less communication: recent advances. SIGACT News **52**(1), 71–80 (2021). https://doi.org/10.1145/3457588.3457600
13. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: HotStuff: BFT consensus with linearity and responsiveness. In: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, pp. 347–356 (2019)
14. The Diem Team: DiemBFT v4: State Machine Replication in the Diem Blockchain (2021). https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf
15. Castro, M., Liskov, B.: Practical Byzantine fault tolerance. ACM Trans. Comput. Syst. **20**, 359–368 (2002)
16. Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus, pp. 1–14 (2018). arXiv:1807.04938
17. Dolev, D., Strong, H.R.: Authenticated algorithms for Byzantine agreement. SIAM J. Comput. **12**(4), 656–666 (1983)
18. Abraham, I., Devadas, S., Nayak, K., Ren, L.: Brief announcement: practical synchronous byzantine consensus. In: Richa, A.W. (ed.) 31st International Symposium on Distributed Computing, DISC 2017, 16–20 Oct, 2017, Vienna, Austria. LIPIcs, vol. 91, pp. 41–1414. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.DISC.2017.41
19. Locher, T.: Fast Byzantine agreement for permissioned distributed ledgers. In: Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 371–382 (2020)
20. Micali, S.: Byzantine agreement, made trivial (2017). https://people.csail.mit.edu/silvio/Selected%20Scientific%20Papers/Distributed%20Computation/BYZANTYNE%20AGREEMENT%20MADE%20TRIVIAL.pdf

21. Ben-Or, M.: Another advantage of free choice: completely asynchronous agreement protocols. In: Proceedings of the Second Annual Symposium on Principles of Distributed Computing, pp. 27–30 (1983)
22. Abraham, I., Malkhi, D., Spiegelman, A.: Asymptotically optimal validated asynchronous Byzantine agreement. In: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, pp. 337–346 (2019)
23. Lu, Y., Lu, Z., Tang, Q., Wang, G.: Dumbo-MVBA: optimal multi-valued validated asynchronous byzantine agreement, revisited. In: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, pp. 129–138 (2020)
24. Abraham, I., Chan, T.H.H., Dolev, D., Nayak, K., Pass, R., Ren, L., Shi, E.: Communication complexity of Byzantine agreement, revisited. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. PODC '19, pp. 317–326. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293611.3331629
25. Chen, J., Gorbunov, S., Micali, S., Vlachos, G.: Algorand agreement: super fast and partition resilient Byzantine agreement. Cryptology ePrint Archive, pp. 1–10 (2018)
26. Cohen, S., Keidar, I., Spiegelman, A.: Brief announcement: not a COINcidence: sub-quadratic asynchronous Byzantine agreement WHP. In: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, pp. 175–177 (2020)
27. King, V., Saia, J.: Breaking the $O(n^2)$ bit barrier: scalable Byzantine agreement with an adaptive adversary. J. ACM **58**(4), 1–24 (2011)
28. Libert, B., Joye, M., Yung, M.: Born and raised distributively: fully distributed non-interactive adaptively-secure threshold signatures with short shares. Theor. Comput. Sci. **645**, 1–24 (2016)
29. Abraham, I., Ben-David, N., Yandamuri, S.: Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement. In: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, vol. 1, pp. 381–391 (2022). https://doi.org/10.1145/3519270.3538426
30. Mostéfaoui, A., Moumen, H., Raynal, M.: Signature-free asynchronous binary Byzantine consensus with t < n/3, O(n2) messages, and O(1) expected time. J. ACM **62**, 13121 (2015). https://doi.org/10.1145/2785953
31. de Souza, L.F., Kuznetsov, P., Tonkikh, A.: Distributed randomness from approximate agreement. In: Scheideler, C. (ed.) 36th International Symposium on Distributed Computing, DISC 2022, 25–27 Oct, 2022, Augusta, Georgia, USA. LIPIcs, vol. 246, pp. 24–12421. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.DISC.2022.24
32. Choudhury, A., Patra, A.: On the communication efficiency of statistically secure asynchronous MPC with optimal resilience. J. Cryptol. **36**(2), 13 (2023). https://doi.org/10.1007/s00145-023-09451-9
33. Bracha, G.: Asynchronous Byzantine agreement protocols. Inf. Comput. **75**(2), 130–143 (1987). https://doi.org/10.1016/0890-5401(87)90054-X
34. Andrychowicz, M., Dziembowski, S.: PoW-based distributed cryptography with no trusted setup. In: Gennaro, R., Robshaw, M. (eds.) Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, 16–20 Aug, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9216, pp. 379–399. Springer (2015). https://doi.org/10.1007/978-3-662-48000-7_19
35. Garay, J.A., Kiayias, A., Leonardos, N., Panagiotakos, G.: Bootstrapping the blockchain, with applications to consensus and fast PKI setup. In: Abdalla, M., Dahab, R. (eds.) Public-Key Cryptography - PKC 2018 - 21st IACR International Conference on Practice and Theory of Public-Key Cryptography, Rio de Janeiro, Brazil, 25–29 Mar, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10770, pp. 465–495. Springer (2018). https://doi.org/10.1007/978-3-319-76581-5_16
36. Abraham, I., Jovanovic, P., Maller, M., Meiklejohn, S., Stern, G., Tomescu, A.: Reaching consensus for asynchronous distributed key generation. In: Miller, A., Censor-Hillel, K., Korhonen, J.H. (eds.) PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, 26–30 July, 2021, pp. 363–373. ACM (2021). https://doi.org/10.1145/3465084.3467914
37. Cachin, C., Kursawe, K., Shoup, V.: Random oracles in Constantinople: practical asynchronous Byzantine agreement using cryptography. J. Cryptol. **18**(3), 219–246 (2005). https://doi.org/10.1007/s00145-005-0318-0
38. Rabin, M.O.: Randomized Byzantine generals. In: 24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7–9 Nov, 1983, pp. 403–409. IEEE Computer Society (1983). https://doi.org/10.1109/SFCS.1983.48
39. Gafni, E.: Round-by-round fault detectors: unifying synchrony and asynchrony. In: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, pp. 143–152 (1998)
40. Guerraoui, R., Raynal, M.: The information structure of indulgent consensus. IEEE Trans. Comput. **53**(4), 453–466 (2004)
41. Keidar, I., Shraer, A.: Timeliness, failure-detectors, and consensus performance. In: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, vol. 2006, pp. 169–178 (2006)
42. Golan Gueta, G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M., Seredinschi, D.A., Tamir, O., Tomescu, A.: SBFT: a scalable and decentralized trust infrastructure. In: Proceedings - 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, pp. 568–580 (2019)
43. Martin, J.P., Alvisi, L.: Fast Byzantine consensus. In: Proceedings of the International Conference on Dependable Systems and Networks, pp. 402–411 (2005)
44. Ramasamy, H.G.V., Cachin, C.: Parsimonious asynchronous Byzantine-fault-tolerant atomic broadcast. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 3974 LNCS, pp. 88–102 (2006)
45. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative Byzantine fault tolerance. ACM Trans. Comput. Syst. **27**(4), 1–39 (2009)
46. Kuznetsov, P., Tonkikh, A., Zhang, Y.X.: Revisiting optimal resilience of fast Byzantine consensus. In: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC), vol. 1, pp. 343–353 (2021)
47. Pass, R., Shi, E.: Thunderella: blockchains with optimistic instant confirmation. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 10821 LNCS, pp. 3–33 (2018)
48. Naor, O., Baudet, M., Malkhi, D., Spiegelman, A.: Cogsworth: Byzantine view synchronization. Cryptoeconomic systems (2021)
49. Naor, O., Keidar, I.: Expected linear round synchronization: the missing link for linear Byzantine SMR. In: 34th International Symposium on Distributed Computing (DISC), vol. 179 (2020)
50. Bravo, M., Chockler, G., Gotsman, A.: Making Byzantine consensus live. In: 34th International Symposium on Distributed Computing (DISC), vol. 179, pp. 1–17 (2020)
51. Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. In: Proceedings of the 10th ACM Symposium on Principles of Distributed Computing, pp. 225–267 (1996)
52. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. In: Proceedings of the Annual ACM Symposium on Principles of Distributed Computing, vol. 43, pp. 147–158 (1992)
53. Kihlstrom, K.P., Moser, L.E., Melliar-Smith, P.M.: Byzantine fault detectors for solving consensus. Comput. J. **46**(1), 16–35 (2003)

54. Antoniadis, K., Desjardins, A., Gramoli, V., Guerraoui, R., Zablotchi, I.: Leaderless consensus. In: Proceedings - International Conference on Distributed Computing Systems, vol. 2021, pp. 392–402 (2021)
55. Dolev, D., Halpern, J.Y., Simons, B., Strong, R.: Dynamic fault-tolerant clock synchronization. J. ACM (JACM) **42**(1), 143–185 (1995)
56. Srikanth, T.K., Toueg, S.: Optimal clock synchronization. J. Assoc. Comput. Mach. **34**(3), 71–86 (1987)
57. Lewis-Pye, A.: Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model (2022). arXiv arXiv:2201.01107
58. Abraham, I., Gueta, G., Malkhi, D.: Hot-stuff the linear, optimal-resilience, one-message BFT devil. *CoRR* arXiv:1803.05069 (2018)
59. Civit, P., Gilbert, S., Guerraoui, R., Komatovic, J., Vidigueira, M.: Strong Byzantine agreement with adaptive word complexity (2023). *CoRR* arXiv:2308.03524
60. Cohen, S., Keidar, I., Spiegelman, A.: Make every word count: adaptive Byzantine agreement with fewer words. In: Hillel, E., Palmieri, R., Rivière, E. (eds.) 26th International Conference on Principles of Distributed Systems, OPODIS 2022, 13–15 Dec, 2022, Brussels, Belgium. LIPIcs, vol. 253, pp. 18–11821. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.OPODIS.2022.18
61. Civit, P., Gilbert, S., Guerraoui, R., Komatovic, J., Monti, M., Vidigueira, M.: Every bit counts in consensus (2023). arXiv preprint arXiv:2306.00431