# Byzantine Consensus is $\Theta(n^2)$

## The Dolev-Reischuk Bound is Tight even in Partial Synchrony!

**Pierre Civit**
Sorbonne University, France

**Muhammad Ayaz Dzulfikar**
NUS Singapore, Singapore

**Seth Gilbert**
NUS Singapore, Singapore

**Vincent Gramoli**
University of Sydney and Redbelly Network, Australia

**Rachid Guerraoui**
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

**Jovan Komatovic**
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

**Manuel Vidigueira**
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

**Abstract**

The Dolev-Reischuk bound says that any deterministic Byzantine consensus protocol has (at least) quadratic communication complexity in the worst case. 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. 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 quadratic worst-case communication complexity. In addition, SQuad is optimally-resilient and achieves linear 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 quadratic communication complexity and linear latency complexity, which we utilize in order to obtain SQuad.

**2012 ACM Subject Classification** Theory of computation → Distributed algorithms

**Keywords and phrases** Optimal Byzantine consensus, Communication complexity, Latency complexity

**Digital Object Identifier** 10.4230/LIPIcs.DISC.2022.11

**Related Version** The full version of this paper, which includes detailed proofs, is available online [19].
*Full Version*: `https://arxiv.org/abs/2208.09262`

**Funding** *Seth Gilbert*: Supported in part by Singapore MOE grant MOE2018-T2-1-160.

## 1 Introduction

Byzantine consensus [38] is a fundamental distributed computing problem. In recent years, it has become the target of widespread attention due to the advent of blockchain [22, 4, 31] and decentralized cloud computing [41], 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 processes follow a prescribed


36th International Symposium on Distributed Computing (DISC 2022).
Editor: Christian Scheideler; Article No. 11; pp. 11:1–11:19
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

protocol, whereas faulty processes (up to $f > 0$) can arbitrarily deviate from it. Each correct process *proposes* a value, and should eventually *decide* a value. 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 [25] says that any deterministic solution of the Byzantine consensus problem requires correct processes to exchange (at least) a quadratic number of bits of information. It has been shown that the bound is tight in synchronous environments [10, 46]. However, for the partially synchronous environments [26] in which the network becomes synchronous only after some unknown Global Stabilization Time ($GST$), no Byzantine consensus protocol achieving quadratic communication complexity is known.[1] Therefore, the question remains whether a partially synchronous Byzantine consensus with quadratic communication complexity exists [20]. Until now, the most efficient known solutions in partially synchronous environments had cubic communication complexity (e.g., HotStuff [56], binary DBFT [22]).

We close the gap by introducing SQuad, a partially synchronous Byzantine consensus protocol with quadratic worst-case communication complexity, matching the Dolev-Reischuk [25] bound. In addition, SQuad is optimally-resilient and achieves optimal linear worst-case latency.

**Partially synchronous "leader-based" Byzantine consensus.** Partially synchronous "leader-based" consensus protocols [56, 55, 15, 13] 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. In summary, the communication complexity of such protocols can be approximated by $n \cdot C + S$, where:

- $C$ denotes the maximum number of bits 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, 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 [56] 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 $O(n)$ bits in a view if it is the leader of the view, and $O(1)$ bits 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-

[1] No deterministic protocol solves Byzantine consensus in a completely asynchronous environment [27].

all" communication exploited by its view synchronizer in *every* view.[2] 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 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.

**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 quadratic communication complexity in partially synchronous environments.

**RareSync.** The main technical contribution of this work is RareSync, a partially synchronous view synchronizer that achieves synchronization within $O(f)$ 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 [55]), 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 $\delta$ 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 $\Delta$-time overlap. In brief, RareSync uses communication to synchronize processes, while relying on local timeouts (and not communication!) to keep them synchronized.

**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) $O(f)$ 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 the 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 $O(f)$ time after $GST$.

**Roadmap.** We discuss related work in §2. In §3, we define the system model. We introduce RareSync in §4. In §5, we present SQuad. We conclude the paper in §6.

[2] While HotStuff [56] does not explicitly state how the view synchronization is achieved, we have that $S = O(n^3)$ in Diem BFT [55], which is a mature implementation of the HotStuff protocol.

## 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.

**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 [46]. Optimality follows from two lower bounds: Dolev and Reischuk show that any Byzantine consensus protocol has an execution with quadratic communication complexity [25]; Dolev and Strong show that any synchronous Byzantine consensus protocol has an execution with $f + 1$ rounds [23]. Various other works have tackled the problem of minimizing the latency of Byzantine consensus [2, 42, 45].

**Randomization.** A classical approach to circumvent the FLP impossibility [27] is using randomization [9], where termination is not ensured deterministically. Exciting recent results by Abraham *et al.* [5] and Lu *et al.* [43] 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 [53] 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 randomized 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 [1, 6]. (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 [18, 21, 35].

**Authentication.** Most of the results above are *authenticated*: they assume a trusted setup phase[3] 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 [5, 43]) rely on *threshold signatures* (such as [40]). The Dolev-Reischuk [25] 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 [22] achieves $O(n^3)$ communication complexity. For randomized non-authenticated Byzantine agreement protocols, Mostefaoui *et al.* [47] achieve $O(n^2)$ communication complexity—but they assume a perfect common coin, for which efficient implementations may also require signatures.

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) [11, 7, 29], (2) set up a threshold scheme [3] without a *trusted dealer*, and (3) asynchronously emulate a perfect common coin [14] used by randomized Byzantine consensus protocols [51, 47, 5, 43].

**Other related work.** In this paper, we focus on the partially synchronous setting [26], 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 [28, 32, 33] that were developed to express partially synchronous protocols using a round-based paradigm. More specifically, state-of-the-art partially synchronous BFT protocols [55, 13, 56, 30] have been developed

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

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 [44, 52, 36, 37, 50], 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 [30] 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 optimistic scenarios [48] and in terms of *expected* complexity [49]. The problem has been formalized more precisely in [12] 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 $\Omega$ [16, 17], capturing the weakest failure detection information needed to solve consensus (and extended to the Byzantine context in [34]). Leaderless partially synchronous Byzantine consensus protocols have also been proposed [8], somehow indicating that the notion of a leader is not necessary in the mechanisms of a consensus protocol, even if $\Omega$ is the weakest failure detector needed to solve the problem. Clock synchronization [24, 54] and view synchronization are orthogonal problems.

**Concurrent research.** We have recently discovered concurrent and independent research by Lewis-Pye [39]. 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 [39] for a different presentation of a similar approach.

## 3 System Model

**Processes.** We consider a static set $\{P_1, P_2, ..., P_n\}$ 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.

**Partial synchrony.** We consider the partially synchronous model introduced in [26]. For every execution, there exists a Global Stabilization Time ($GST$) and a positive duration $\delta$ such that message delays are bounded by $\delta$ after $GST$. Furthermore, $GST$ is not known to processes, whereas $\delta$ 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.

**Cryptographic primitives.** We assume a $(k, n)$-threshold signature scheme [40], 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 a partial signature of a message $m$ by invoking $\mathit{ShareSign}_i(m)$. A partial signature *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 $Combine(S)$; a combined signature $tcombined$ of message $m$ can be verified by $CombinedVerify(m, tcombined)$. Where appropriate, invocations of $ShareVerify(\cdot)$ and $CombinedVerify(\cdot)$ are implicit in our descriptions of protocols. P_Signature and T_Signature denote a partial signature and a (combined) threshold signature, respectively.

**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 $\alpha$.

A *word* contains a constant number of signatures and values. Each message contains at least a single word. We define the communication complexity of $\alpha$ 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 $\alpha$ is 0. The latency complexity of $\alpha$ is $\max(0, t_d(\alpha) - GST)$.

The *communication complexity* of Consensus is defined as

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

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

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

We underline that the number of words sent by correct processes before $GST$ is unbounded in any partially synchronous Byzantine consensus protocol [53]. Moreover, not a single correct process is guaranteed to decide before $GST$ in any partially synchronous Byzantine consensus protocol [27]; that is why the latency complexity of such protocols is measured from $GST$.

# 4 RareSync

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

## 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 [12, 49, 48]. More precisely, let View $= \{1, 2, ...\}$ denote the set of views. For each view $v \in$ View, we define 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 advance$(v)$ indication occurs at time $t$. Moreover, a correct process *is in view* $v$ between the time $t$ (including $t$) at which the advance$(v)$ indication occurs and the time $t'$ (excluding $t'$) at which the next advance$(v' \neq v)$ indication occurs. If an 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) $\Delta$ 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)* leader$(v)$ *is correct.*

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

**Complexity of view synchronization.** Let Synchronizer be a partially synchronous view synchronizer and let $\mathcal{E}(\mathsf{Synchronizer})$ denote the set of all possible executions. Let $\alpha \in \mathcal{E}(\mathsf{Synchronizer})$ be an execution and $t_s(\alpha)$ be the first synchronization time at or after $GST$ in $\alpha$ ($t_s(\alpha) \geq GST$). We define the communication complexity of $\alpha$ 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 $\alpha$ is $t_s(\alpha)+\Delta-GST$.

The *communication complexity* of Synchronizer is defined as

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

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

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

## 4.2 Protocol

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

**Intuition.** We group views into *epochs*, where each epoch contains $f+1$ sequential views; $\mathsf{Epoch} = \{1, 2, ...\}$ 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 1 illustrates the main idea behind RareSync.

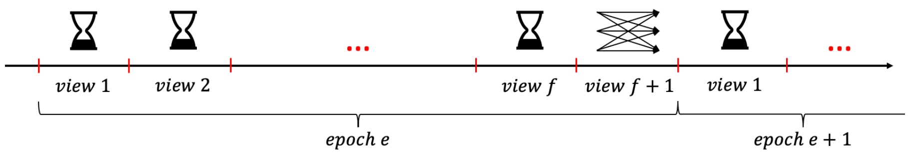


**Figure 1** Intuition behind RareSync: Processes communicate only in the last view of an epoch; before the last view, they rely solely on local timeouts.

Roughly speaking, after $GST$, all correct processes simultaneously enter the same epoch within $O(f)$ time. After entering the same epoch, processes are guaranteed to synchronize in that epoch, which takes (at most) an additional $O(f)$ time. Thus, the latency complexity of RareSync is $O(f)$. 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 $GST$.

**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 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 $\delta$ time (either line 18 or line 24). This $\delta$-waiting step is introduced to limit the number of epochs $P_i$ can enter within any $\delta$ 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 $\delta$ after $GST$, we force a process to wait $\delta$ time, during which it receives all accumulated messages, before entering the largest known epoch.

Finally, after $\delta$ 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 Correctness and Complexity: Proof Sketch

This subsection presents a proof sketch 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 $O(f)$. This is done by proving that synchronization happens within (at most) 2 epochs after $GST$. As for the communication complexity, we prove that any

**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 $epoch\_sig_i \leftarrow \bot$ ▷ 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

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 $O(n^2) = O(1) \cdot O(n) \cdot n$. 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 $e_{\mathit{final}} = e_{max}+1$. 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, $e_{max}=0$.*[4]

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

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

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 $e_{\mathit{final}}$ by time $t_{e_{\mathit{final}}}+2\delta$.*

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

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

By Result 3, no future epoch can be entered before time $t_{e_{\mathit{final}}} + epoch\_duration$. This is precisely enough time for the first correct process (the one to enter $e_{\mathit{final}}$ at $t_{e_{\mathit{final}}}$) to go through all $f+1$ views of $e_{\mathit{final}}$, 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 $2\delta$ between processes (Result 4) applies to every view in $e_{\mathit{final}}$. Thus, all correct processes overlap with each other for (at least) $view\_duration - 2\delta = \Delta$ time in every view of $e_{\mathit{final}}$. 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 $e_{\mathit{final}}$, i.e., there is a synchronization time $t_s$ such that $t_{e_{\mathit{final}}} + \Delta \leq t_s + \Delta \leq t_{e_{\mathit{final}}} + epoch\_duration$.

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

[4] Epoch 0 is considered as a special epoch. Note that $0 \notin \mathsf{Epoch}$, where $\mathsf{Epoch}$ denotes the set of epochs (see §4.2).

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 $e_{max}$.[5] 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 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 2 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 $O(f)$.*

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, $t_s + \Delta - GST \leq 2epoch\_duration + 4\delta = O(f)$.

**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 $\delta$, 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 $O(n^2) = O(1) \cdot O(n) \cdot n$.

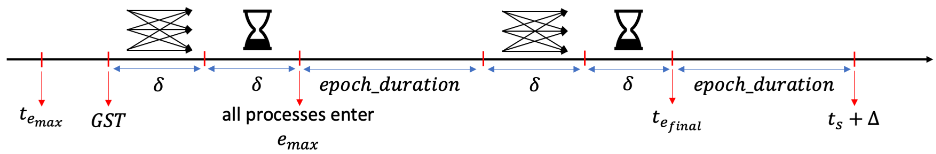


**Figure 2** Worst-case latency of RareSync: $t_s + \Delta - GST \leq 2epoch\_duration + 4\delta$.

▶ **Theorem 2.** *RareSync is a partially synchronous view synchronizer with (1) $O(n^2)$ communication complexity, and (2) $O(f)$ latency complexity.*

## 5 SQuad

This section introduces SQuad, a partially synchronous Byzantine consensus protocol with optimal resilience [26]. SQuad simultaneously achieves (1) $O(n^2)$ communication complexity, matching the Dolev-Reischuk bound [25], and (2) $O(f)$ latency complexity, matching the Dolev-Strong bound [23].

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

First, we present QUAD, a partially synchronous Byzantine consensus protocol ensuring weak validity (§5.1). QUAD achieves quadratic communication complexity and linear latency complexity. Then, we construct SQUAD by adding a simple preprocessing phase to QUAD (§5.2).

## 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 [25] does not apply to Byzantine protocols satisfying weak validity; hence, we do not know whether QUAD has optimal communication complexity. As explained in §5.2, we accompany QUAD by a preprocessing phase to obtain SQUAD.

QUAD (Algorithm 3) uses the same view core module as HotStuff [56], 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 §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 $O(f)$ latency complexity.

**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 [56].

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 $\sigma$ that "enough" processes have supported its "prepare-phase" value; $\sigma$ is a threshold signature. Then, it disseminates $\sigma$ to all processes via a PRECOMMIT message. Once a process receives the PRECOMMIT message carrying $\sigma$, it updates its prepare QC to $\sigma$ 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 $\sigma'$ that "enough" processes have adopted its "precommit-phase" value (by updating their prepare QC); $\sigma'$ is a threshold signature. Then, it disseminates $\sigma'$ to all processes via a COMMIT message. Once a process receives the COMMIT message carrying $\sigma'$, it updates its locked QC to $\sigma'$ 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 $\sigma''$, and relays $\sigma''$ to all processes via a DECIDE message. When a process receives the DECIDE message carrying $\sigma''$, it decides the value associated with $\sigma''$.

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).

**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 $8\delta$ 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).

**Algorithm 3** Quad: Pseudocode (for process $P_i$)

```
1:  Modules:
2:      View_Core core
3:      View_Synchronizer synchronizer ← RareSync(Δ = 8δ)
4:  upon init(Value proposal):                          ▷ propose value proposal
5:      core.init(proposal)                ▷ initialize the view core with the proposal
6:      synchronizer.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 value decision
```

**Proof sketch.** 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 synchronization property, this eventually happens, which implies that Quad satisfies termination. As processes synchronize within $O(f)$ time after $GST$, the latency complexity of Quad is $O(f)$.

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.

▶ **Theorem 3.** *Quad is a Byzantine consensus protocol ensuring weak validity with (1) $O(n^2)$ communication complexity, and (2) $O(f)$ latency complexity.*

## 5.2 SQuad: Protocol Description

At last, we present SQuad, which we derive from Quad.

**Deriving SQuad from Quad.** Imagine a locally-verifiable, *constant-sized* cryptographic proof $\sigma_v$ 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 §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.

**Certification phase.** SQuad utilizes its *certification phase* (Algorithm 4) to obtain the introduced constant-sized cryptographic proofs; we call these proofs *certificates*.[6] Formally, $\mathsf{Certificate}$ denotes the 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 $\sigma_v$ 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 §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.

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

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).

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

**Algorithm 4** 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 σ_v ← Combine({sig | sig is received in a DISCLOSE message})
7:     broadcast ⟨CERTIFICATE, v, σ_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 σ_⊥ ← Combine({sig | sig is received in an ALLOW-ANY message})
15:    broadcast ⟨CERTIFICATE, ⊥, σ_⊥⟩                                   ▷ disseminate the certificate
16:    exit the certification phase
17: ▷ a certificate for v is obtained; v can be ⊥, meaning that σ_v vouches for any value
18: upon reception of ⟨CERTIFICATE, Value v, Certificate σ_v⟩:
19:    broadcast ⟨CERTIFICATE, v, σ_v⟩                                   ▷ disseminate the certificate
20:    exit the certification phase
21: function verify(Value v, Certificate σ):
22:    if CombinedVerify("any value", σ) = true: return true
23:    else if CombinedVerify(v, σ) = true: return true
24:    else return false
```

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 4) 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.

**SQuad = Certification phase + Quad.** We obtain SQuad by combining the certification phase with Quad. The pseudocode of SQuad is given in Algorithm 5.

**Algorithm 5** 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 σ_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, σ_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 $\sigma_v$ for a value $v$, it proposes $(v, \sigma_v)$ 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).

▶ **Theorem 4.** *SQuad is a Byzantine consensus protocol with (1) $O(n^2)$ communication complexity, and (2) $O(f)$ latency complexity.*

## 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 $O(f)$ latency complexity. SQuad owes its complexity to RareSync, an “epoch-based” view synchronizer ensuring synchronization with quadratic communication and linear latency in partial synchrony.

### Acknowledgments

The authors would like to thank Gregory Chockler and Alexey Gotsman for helpful conversations. This work is supported in part by the ARC Future Fellowship funding scheme (#180100496).

## References

1 Ittai Abraham, T-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication Complexity of Byzantine Agreement, Revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC ’19, page 317–326, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3293611.3331629`.

2 Ittai Abraham, Srinivas Devadas, Kartik Nayak, and Ling Ren. Brief Announcement: Practical Synchronous Byzantine Consensus. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, volume 91 of *LIPIcs*, pages 41:1–41:4. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.DISC.2017.41.

3 Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, Gilad Stern, and Alin Tomescu. Reaching Consensus for Asynchronous Distributed Key Generation. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 363–373. ACM, 2021. doi:10.1145/3465084.3467914.

4 Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus. In James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão, editors, *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, volume 95 of *LIPIcs*, pages 25:1–25:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

5 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

6 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 337–346, 2019.

7 Marcin Andrychowicz and Stefan Dziembowski. PoW-Based Distributed Cryptography with No Trusted Setup. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 379–399. Springer, 2015. doi:10.1007/978-3-662-48000-7_19.

8 Karolos Antoniadis, Antoine Desjardins, Vincent Gramoli, Rachid Guerraoui, and Igor Zablotchi. Leaderless Consensus. In *Proceedings - International Conference on Distributed Computing Systems*, volume 2021-July, pages 392–402, 2021.

9 Michael Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols. *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 27–30, 1983.

10 Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Bit Optimal Distributed Consensus. *Computer Science: Research and Applications*, page 313–321, 1992.

11 Gabriel Bracha. Asynchronous Byzantine Agreement Protocols. *Inf. Comput.*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

12 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making Byzantine Consensus Live. In *34th International Symposium on Distributed Computing (DISC)*, volume 179, pages 1–17, 2020.

13 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. pages 1–14, 2018. URL: https://arxiv.org/pdf/1807.04938.pdf, arXiv:1807.04938.

14 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *J. Cryptol.*, 18(3):219–246, 2005. doi:10.1007/s00145-005-0318-0.

15 Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. *ACM Trans. Comput. Syst.*, (February):359–368, 2002.

16 Tushar Chandra and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, (2):225–267, 1996.

17 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, 43(4):147–158, 1992.

18 Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. Algorand Agreement: Super Fast and Partition Resilient Byzantine Agreement. *Cryptology ePrint Archive*, 377:1–10, 2018. URL: https://eprint.iacr.org/2018/377.pdf.

19 Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. Byzantine Consensus is $\Theta(n^2)$: The Dolev-Reischuk Bound is Tight

even in Partial Synchrony! [Extended Version], 2022. URL: `https://arxiv.org/abs/2208.09262`, `doi:10.48550/ARXIV.2208.09262`.

20 Shir Cohen, Idit Keidar, and Oded Naor. Byzantine Agreement with Less Communication: Recent Advances. *SIGACT News*, 52(1):71–80, 2021. `doi:10.1145/3457588.3457600`.

21 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Brief Announcement: Not a COINcidence: Sub-Quadratic Asynchronous Byzantine Agreement WHP. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 175–177, 2020.

22 Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. 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}*, pages 1–41, 2017. `arXiv:1702.03068`.

23 D. Dolev and H. R. Strong. Authenticated Algorithms for Byzantine Agreement. 12(4):656–666, 1983.

24 Danny Dolev, Joseph Y. Halpern, Barbara Simons, and Ray Strong. Dynamic Fault-Tolerant Clock Synchronization. *Journal of the ACM (JACM)*, 42(1):143–185, 1995.

25 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM (JACM)*, 1985.

26 Cynthia Dwork, Lynch Nancy, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

27 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the Association for Computing Machinery,*, 32(2):374–382, 1985.

28 Eli Gafni. Round-by-Round Fault Detectors: Unifying Synchrony and Asynchrony. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 143–152, 1998.

29 Juan A. Garay, Aggelos Kiayias, Nikos Leonardos, and Giorgos Panagiotakos. Bootstrapping the Blockchain, with Applications to Consensus and Fast PKI Setup. In Michel Abdalla and Ricardo Dahab, editors, *Public-Key Cryptography - PKC 2018 - 21st IACR International Conference on Practice and Theory of Public-Key Cryptography, Rio de Janeiro, Brazil, March 25-29, 2018, Proceedings, Part II*, volume 10770 of *Lecture Notes in Computer Science*, pages 465–495. Springer, 2018. `doi:10.1007/978-3-319-76581-5\_16`.

30 Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A Scalable and Decentralized Trust Infrastructure. *Proceedings - 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019*, pages 568–580, 2019.

31 Vincent Gramoli. From blockchain consensus back to Byzantine consensus. *Future Gener. Comput. Syst.*, 107:760–769, 2020.

32 Rachid Guerraoui and Michel Raynal. The Information Structure of Indulgent Consensus. *{IEEE} Trans. Computers*, 53(4):453–466, 2004.

33 Idit Keidar and Alexander Shraer. Timeliness, Failure-Detectors, and Consensus Performance. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, 2006:169–178, 2006.

34 Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Byzantine Fault Detectors for Solving Consensus. *The Computer Journal*, 46(1):16–35, 2003.

35 Valerie King and Jared Saia. Breaking the $O(n^2)$ Bit Barrier: Scalable Byzantine agreement with an Adaptive Adversary. *Journal of the ACM*, 58(4):1–24, 2011.

36 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems*, 27(4), 2009.

37 Petr Kuznetsov, Andrei Tonkikh, and Yan X. Zhang. Revisiting Optimal Resilience of Fast Byzantine Consensus. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1(1):343–353, 2021.

38 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

39 Andrew Lewis-Pye. Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model, 2022. URL: `https://arxiv.org/abs/2201.01107`, `doi:10.48550/ARXIV.2201.01107`.

40 Benoît Libert, Marc Joye, and Moti Yung. Born and Raised Distributively: Fully Distributed Non-Interactive Adaptively-Secure Threshold Signatures with Short Shares. *Theoretical Computer Science*, 645:1–24, 2016.

41 JongBeom Lim, Taeweon Suh, Joon-Min Gil, and Heon-Chang Yu. Scalable and leaderless Byzantine consensus in cloud computing environments. *Inf. Syst. Frontiers*, 16(1):19–34, 2014.

42 Thomas Locher. Fast Byzantine Agreement for Permissioned Distributed Ledgers. *Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 371–382, 2020.

43 Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-MVBA: Optimal Multi-Valued Validated Asynchronous Byzantine Agreement, Revisited. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 129–138, 2020.

44 Jean Philippe Martin and Lorenzo Alvisi. Fast Byzantine Consensus. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 402–411, 2005.

45 Silvio Micali. Byzantine Agreement , Made Trivial. 2017.

46 Atsuki Momose and Ling Ren. Optimal Communication Complexity of Authenticated Byzantine Agreement. In *35th International Symposium on Distributed Computing (DISC)*, volume 209, pages 32:1–32:0. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany, 2021.

47 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-Free Asynchronous Binary Byzantine Consensus with t < n/3, O(n2) Messages, and O(1) Expected Time. *J. ACM*, 62(4):31:1–31:21, 2015. doi:10.1145/2785953.

48 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine View Synchronization. *Cryptoeconomic Systems*, 2021.

49 Oded Naor and Idit Keidar. Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR. *34th International Symposium on Distributed Computing (DISC)*, 179, 2020.

50 Rafael Pass and Elaine Shi. Thunderella: Blockchains with Optimistic Instant Confirmation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10821 LNCS:3–33, 2018.

51 Michael O. Rabin. Randomized Byzantine Generals. In *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*, pages 403–409. IEEE Computer Society, 1983. doi:10.1109/SFCS.1983.48.

52 Hari Govind V. Ramasamy and Christian Cachin. Parsimonious Asynchronous Byzantine-Fault-Tolerant Atomic Broadcast. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3974 LNCS:88–102, 2006.

53 Alexander Spiegelman. In Search for an Optimal Authenticated Byzantine Agreement. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2021/14840, doi:10.4230/LIPIcs.DISC.2021.38.

54 T. K. Srikanth and Sam Toueg. Optimal Clock Synchronization. *Journal of the Association for Computing Machinery*, 34(3):71–86, 1987.

55 The Diem Team. DiemBFT v4: State Machine Replication in the Diem Blockchain, 2021. URL: https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf.

56 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.