In Search for an Optimal Authenticated Byzantine Agreement

Alexander Spiegelman ✉
Novi Research, Menlo Park, USA

Abstract

In this paper, we challenge the conventional approach of state machine replication systems to design deterministic agreement protocols in the eventually synchronous communication model. We first prove that no such protocol can guarantee bounded communication cost before the global stabilization time and propose a different approach that hopes for the best (synchrony) but prepares for the worst (asynchrony). Accordingly, we design an *optimistic* byzantine agreement protocol that first tries an efficient deterministic algorithm that relies on synchrony for termination only, and then, only if an agreement was not reached due to asynchrony, the protocol uses a randomized asynchronous protocol for fallback that guarantees termination with probability 1.

We formally prove that our protocol achieves optimal communication complexity under all network conditions and failure scenarios. We first prove a lower bound of $\Omega(ft + t)$ for synchronous deterministic byzantine agreement protocols, where t is the failure threshold, and f is the actual number of failures. Then, we present a tight upper bound and use it for the synchronous part of the optimistic protocol. Finally, for the asynchronous fallback, we use a variant of the (optimal) VABA protocol, which we reconstruct to safely combine it with the synchronous part.

We believe that our adaptive to failures synchronous byzantine agreement protocol has an independent interest since it is the first protocol we are aware of which communication complexity optimally depends on the actual number of failures.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Security and privacy → Distributed systems security

Keywords and phrases Byzantine agreement, Optimistic, Asynchronous fallback

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.38

Related Version *Full Version*: `https://arxiv.org/pdf/2002.06993.pdf`

1 Introduction

With the emergence of the Blockchain use case, designing efficient geo-replicated Byzantine tolerant state machine replication (SMR) systems is now one of the most challenging problems in distributed computing. The core of every Byzantine SMR system is the Byzantine agreement problem (see [3] for a survey), which was first introduced four decades ago [33] and has been intensively studied since then [11, 22, 26, 24]. The bottleneck in geo-replicated SMR systems is the network communication, and thus a substantial effort in recent years was invested in the search for an optimal communication Byzantine agreement protocol [20, 38, 10, 30].

To circumvent the FLP [17] result that states that deterministic asynchronous agreement protocols are impossible, most SMR solutions [12, 20, 38, 23] assume eventually synchronous communication models and provide safety during asynchronous periods but can guarantee progress only after the global stabilization time (GST).

Therefore, it is quite natural that state-of-the-art authenticated Byzantine agreement protocols [20, 38, 10, 30] focus on reducing communication cost after GST, while putting up with the potentially unbounded cost beforehand. For example, Zyzzyva [23] and later SBFT [20] use threshold signatures [34] and collectors to reduce the quadratic cost induced by the all-to-all communication in each view of the PBFT [12] protocol. HotStuff [38] leverages

ideas presented in Tendermint [10] to propose a linear view-change mechanism, and a few follow-up works [30, 31, 9] proposed algorithms for synchronizing parties between views. Some [30, 31] proposed a synchronizer with a linear cost after GST in failure-free runs, while others [9] provided an implementation that guarantees bounded memory even before GST. However, none of the above algorithms bounds the number of views executed before GST, and thus none of them can guarantee a bounded total communication cost.

We argue in this paper that designing agreement algorithms in the eventually synchronous model is not the best approach to reduce the total communication complexity of SMR systems and propose an alternative approach. That is, we propose to forgo the eventually synchronous assumptions and instead optimistically consider the network to be synchronous and immediately switch to randomized asynchronous treatment if synchrony assumption does not hold. Our goal in this paper is to develop an *optimistic* protocol that adapts to network conditions and actual failures to guarantee termination with an optimal communication cost under all failure and network scenarios.

1.1 Contribution

Vulnerability of the eventually synchronous model. A real network consists of synchronous and asynchronous periods. From a practical point of view, if the synchronous periods are too short, no deterministic Agreement algorithm can make progress [17]. Therefore, to capture the assumption that eventually there will be a long enough synchronous period for a deterministic Agreement to terminate, the eventually synchronous model assumes that every execution has a point, called GST, after which the network is synchronous. In our first result, we capture the inherent vulnerability of algorithms designed in the eventually synchronous communication model. That is, we exploit the fact that GST can occur after an arbitrarily long time to prove the following lower bound:

▶ **Theorem 1.** *There is no eventually synchronous deterministic Byzantine agreement protocol that can tolerate a single failure and guarantee bounded communication cost even in failure-free runs.*

Tight bounds for synchronous Byzantine agreement. To develop an optimal optimistic protocol that achieves optimal communication under all failure and network scenarios we first establish what is the best we can achieve in synchronous settings. Dolev and Reischuk [14] proved that there is no deterministic protocol that solves synchronous Byzantine agreement with $o(t^2)$ communication cost, where t is the failure threshold. We generalize their result by considering the actual number of failures $f \leq t$ and prove the following lower bound:

▶ **Theorem 2.** *Any synchronous deterministic Byzantine agreement protocol has $\Omega(ft + t)$ communication complexity.*

It is important to note that the lower bound holds even for deterministic protocols that are allowed to use perfect cryptographic schemes such as threshold signatures and authenticated links. Then, we present the first deterministic cryptography-based synchronous Byzantine agreement protocol that matches our lower bound for the authenticated case. That is, we prove the following:

▶ **Theorem 3.** *There is a deterministic synchronous authenticated Byzantine agreement protocol with $O(ft + t)$ communication complexity.*

We believe these results are interesting on their own since they are the first to consider the actual number of failures, which was previously considered in the problem of early decision/stopping [15, 21], for communication complexity analysis of the Byzantine agreement problem.

Optimal optimistic Byzantine agreement. Our final contribution is an optimistic Byzantine agreement protocol that tolerates up to $t < n/3$ failures and has asymptotically optimal communication cost under all network conditions and failure scenarios. That is, we prove the following:

▶ **Theorem 4.** *There is an authenticated Byzantine agreement protocol with $O(ft + t)$ communication complexity in synchronous runs and expected $O(t^2)$ communication complexity in all other runs.*

To achieve the result, we combine our optimal adaptive synchronous protocol with an asynchronous fallback, for which we use a variant of VABA [1]. As we shortly explain, the combination is not trivial since we need to preserve safety even if parties decide in different parts of the protocol, and implement an efficient mechanism to prevent honest parties from moving to the fallback in synchronous runs.

1.2 Technical overview

The combination of our synchronous part with the asynchronous fallback introduces two main challenges. The first challenge is to design a mechanism that (1) makes sure parties do not move to the fallback unless necessary for termination, and (2) has $O(ft + t)$ communication complexity in synchronous runs. The difficulty here is twofold: first, parties cannot always distinguish between synchronous and asynchronous runs. Second, they cannot distinguish between honest parties that complain that they did not decide (due to asynchrony) in the first part and Byzantine parties that complain because they wish to increase the communication cost by moving to the asynchronous fallback. To deal with this challenge, we implement a *Help&tryHalting* procedure. In a nutshell, parties try to avoid the fallback part by helping complaining parties learn the decision value and move to the fallback only when the number of complaints indicates that the run is not synchronous. This way, each Byzantine party in a synchronous run cannot increase the communication cost by more than $O(n) = O(t)$, where n is the total number of parties.

The second challenge in the optimistic protocol is to combine both parts in a way that guarantees safety. That is, since some parties may decide in the synchronous part and others in the asynchronous fallback, we need to make sure they decide on the same value. To this end, we use the *leader-based view (LBV)* abstraction, defined in [37], as a building block for both parts. The LBV abstraction captures a single view in a view-by-view agreement protocol such that one of its important properties is that a sequential composition of them preserves safety. For optimal communication cost, we adopt techniques from [38] and [1] to implement the LBV abstraction with an asymptotically linear cost ($O(n)$).

Our synchronous protocol operates up to n sequentially composed pre-defined linear LBV instances, each with a different leader. To achieve an optimal (adaptive to the number of actual failures) cost, leaders invoke their LBVs only if they have not yet decided. In contrast to eventually synchronous protocols, the synchronous part is designed to provide termination only in synchronous runs. Therefore, parties do not need to be synchronized before views, but rather move from one LBV to the next at pre-defined times. As for the asynchronous fallback, we use the linear LBV building block to reconstruct the VABA [1] protocol in a way that forms a sequential composition of LBVs, which in turn allows a convenient sequential composition with the synchronous part.

1.3 Related work

The idea of combining several agreement protocols is not new. The notion of speculative linearizability [19] allows parties to independently switch from one protocol to another, without requiring them to reach agreement to determine the change of a protocol. Aguilera and Toueg [2] presented an hybrid approach to solve asynchronous crash-fault consensus by combining randomization and unreliable failure detection. Guerraoui et al [18] defined an abstraction that captures byzantine agreement protocols and presented a framework to compose several such instances.

Some previous work on Byzantine agreement consider a fallback in the context of the number rounds required for termination [7, 27, 35]. That is, in well-behaved runs parties decide in a single communication round, wheres in all other runs they fallback to a mode that requires more rounds to reach an agreement. We, in contrast, are interested in communication complexity. To the best of our knowledge, our protocol is the first protocol that adapts its communication complexity based on the actual number of failures.

The combination of synchronous and asynchronous runs in the context of Byzantine agreement was previously studied by Blum et al. [5]. Their result is complementary to ours since they deal with optimal resilience rather than optimal communication. They showed lower and upper bounds on the number of failures that both (synchronous and asynchronous) parts can tolerate. For the lower bound, they showed that $t_a + 2t_s < n$, where t_a and t_s is the threshold failure in asynchronous and synchronous runs, respectively. In our protocol $t_a = t_s < n/3$, which means that the protocol is optimal in the sense that neither t_a or t_s can be increased without decreasing the other. For the upper bound, they present a matching algorithm for any t_a and t_s that satisfy the weak validity condition. Our protocol, in contrast, satisfy the more practical external validity condition (see more details in the next section) with an optimal communication cost.

As for asynchronous Byzantine agreement, the lower bound in [1] shows that there is no protocol with optimal resilience and $o(n^2)$ communication complexity. Two recent works by Cohen et al. [13] and Blum et al [4]. circumvent this lower bound by trading optimal resilience. That is, their protocols tolerate $f < (1 - \epsilon)n/3$ Byzantine faults. We consider in this paper optimal resilience and thus our protocol achieves optimal communication complexity in asynchronous runs.

The use of cryptographic tools (e.g. PKI and threshold signatures schemes) is very common in distributed computing to reduce round and communication complexity. To be able to focus on the distributed aspect of the problem, many previous algorithms assume ideal cryptographic tools to avoid the analysis of the small error probability induced by the security parameter. This includes the pioneer protocols for Byzantine broadcast [16, 14] and binary asynchronous Byzantine agreement [6], recent works on synchronous Byzantine agreement [29, 32], and most of the exciting practical algorithms [23, 12] including the state-of-the-art communication efficient ones [12, 38, 20, 10]). We follow this approach and assume ideal threshold signatures schemes for better readability.

2 Model

Following practical solutions [12, 20, 38, 23, 28], we consider a Byzantine message passing peer to peer model with a set Π of n parties and a computationally bounded adversary that corrupts up to $t < n/3$ of them, $O(t) = O(n)$. Parties corrupted by the adversary are called *Byzantine* and may arbitrarily deviate from the protocol. Other parties are *honest*. To strengthen the result we consider an adaptive adversary for the upper bound and static

adversary for the lower bound. The difference is that a *static* adversary must decide what parties to corrupt at the beginning of every execution, whereas an *adaptive* adversary can choose during the executions.

Communication and runs. The communication links are reliable but controlled by the adversary, i.e., all messages sent among honest parties are eventually delivered, but the adversary controls the delivery time. We assume a known to all parameter Δ and say that a run of a protocol is *eventually synchronous* if there is a *global stabilization time (GST)* after which all message sent among honest parties are delivered within Δ time. A run is *synchronous* if GST occurs at time 0, and *asynchronous* if GST never occurs.

The Agreement problem. Each party get an input value from the adversary from some domain $\mathbb{V}$ and the Agreement problem exposes an API to *propose* a value and to output a *decision*. We are interested in protocols that never compromise safety and thus require the following property to be satisfied in all runs:

- Agreement: All honest parties that decide, decide on the same value.

Due to the FLP result [17], no deterministic agreement protocol can provide safety and liveness properties in all asynchronous runs. Therefore, in this paper, we consider protocols that guarantee (deterministic) termination in all synchronous and eventually synchronous runs, and provides a probabilistic termination in asynchronous ones:

- Termination: All honest parties eventually decide.
- Probabilistic-Termination: All honest parties decide with probability 1.

As for validity, honest parties must decide only on values from some domain $\mathbb{V}$. For the lower bounds, to strengthen them as much as possible, we consider the binary case, which is the weakest possible definition:

- Binary validity: The domain of valid values $\mathbb{V} = \{0, 1\}$, and if all honest parties propose the same value $v \in \mathbb{V}$, than no honest party decides on a value other than v.

For the upper bounds, we are interested in practical multi-valued protocols. In contrast to binary validity, in a multi-valued Byzantine agreement we need also to define what is a valid decision in the case that not all parties a priori agree (i.e., propose different values). One option is Weak Validity [33, 5], which allows parties to agree on a pre-defined $\perp$ in that case. This definition is well defined and makes sense for some use cases. When Pease et al. [33] originally defined it, they had in mind a spaceship cockpit with 4 sensors that try to agree even if one is broken (measures a wrong value). However, as Cachin et al, explain in their paper [11] and book [25], this definition is useless for SMR (and Blockchains) since if parties do not a priori agree, then they can keep agreeing on $\perp$ forever leaving the SMR with no "real" progress.

To solve the limitation of being able to agree on $\perp$, we consider the external validity property that was first defined by Cachin et al. [11], which is implicitly or explicitly considered in most practical Byzantine agreement solutions we are aware of [1, 12, 38, 20, 23]. Intuitively, with external validity, parties are allowed to decide on a value proposed by any party (honest and Byzantine) as long as it is valid by some external predicate (e.g., all transaction are valid in the block). To capture the above, we give a formal definition below.

- External validity: The domain of valid values $\mathbb{V}$ is unknown to honest parties. At the beginning of every run, each honest party gets a value v with a proof σ that $v \in \mathbb{V}$ such that all other honest parties can verify.

Note that our definition rules out trivial solutions such as simply deciding on some pre-defined externally valid value because the parties do not know what is externally valid unless they see a proof.

We define an *optimistic Agreement protocol* to be a protocol that guarantees Agreement and External validity in all runs, Termination in all synchronous and eventually synchronous runs, and Probabilistic-Termination in asynchronous runs.

Cryptographic assumptions. We assume a computationally bounded adversary and a trusted dealer that equips parties with cryptographic schemes. Following a common standard in distributed computing and for simplicity of presentation (avoid the analysis of security parameters and negligible error probabilities), we assume that the following cryptographic tools are perfect:

- **Authenticated link.** If an honest party p_i delivers a messages m from an honest party p_j, then p_j previously sent m to p_i.
- **Threshold signatures scheme.** We assume that each party p_i has a private function *share-sign*$_i$, and we assume 3 public functions: *share-validate*, *threshold-sign*, and *threshold-validate*. Informally, given "enough" valid shares, the function *threshold-sign* returns a valid threshold signature. For our algorithm, we sometimes require "enough" to be $t+1$ and sometimes $n-t$. A formal definition is given in the fullpaper [36].

We note that perfect cryptographic schemes do not exist in practice. However, since in real-world systems they often treated as such, we believe that they capture just enough in order to be able to focus on the distributed aspect of the problem. Moreover, all the lower bounds in this paper hold even if protocols can use perfect cryptographic schemes. Thus, the upper bounds are tight in this aspect.

Communication complexity. We denote by f the actual number of corrupted parties in a given run and we are interested in optimistic protocols that utilize f and the network condition to reduce communication cost. Similarly to [1], we say that a *word* contains a constant number of signatures and values, and each message contains at least 1 word. The *communication cost of a run* r is the number of words sent in messages by honest parties in r. For every $0 \leq f \leq t$, let R_f^s and R_f^{es} be the sets of all synchronous and eventually synchronous runs with f corrupted parties, respectively. The *synchronous and eventually synchronous communication cost with* f *failures* is the maximal communication cost of runs in R_f^s and R_f^{es}, respectively. We say that the *synchronous communication cost of a protocol* A is $G(f,t)$ if for every $0 \leq f \leq t$, its synchronous communication cost with f failures is $G(f,t)$. The *asynchronous communication cost of a protocol* A is the expected communication cost of an asynchronous run of A.

3 Lower Bounds

We present two lower bounds on the communication complexity of deterministic Byzantine agreement protocols in synchronous and eventually synchronous runs. For space limitation, the proof of the following lemma appears to Appendix A.

▶ **Theorem 1** (restated). *There is no eventually synchronous deterministic Byzantine agreement protocol that can tolerate a single failure and guarantee bounded communication cost even in failure-free runs.*

We next prove a lower bound that applies even to synchronous Byzantine agreement algorithms and is adaptive to the number of actual failures f. The proof is a generalization of the proof in [14], which has been proved for the Byzantine broadcast problem and considered the worst-case scenario ($f = t$). It is important to note that the proof captures deterministic authenticated algorithms even if they are equipped with perfect cryptographic tools.

The proof of the following Claim is straight forward and for space limitation is omitted.

▷ Claim 5. The synchronous communication cost with 0 failures of any Byzantine agreement algorithm is at least t.

The following Lemma shows that if honest parties send $o(ft)$ messages, then Byzantine parties can prevent honest parties from getting any of them.

▶ **Lemma 6.** *Assume that there is a Byzantine agreement algorithm A, which synchronous communication cost with f failures is $o(ft)$ for some $1 \leq f \leq \lfloor t/2 \rfloor$. Then, for every set $S \subset \Pi$ of f parties and every set of values proposed by honest parties, there is a synchronous run r' s.t. some honest party $p \in S$ does not get any messages in r'.*

Proof. Let $r \in R_f^s$ be a run in which all parties in S are Byzantine that (1) do not send messages among themselves, and (2) ignore all messages they receive and act like honest parties that get no messages. By the assumption, there is a party $p \in S$ that receives less than $t/2$ messages from honest parties in r. Denote the set of (honest) parties outside S that send messages to p in r by $P \subset \Pi \setminus S$ and consider the following run r':

- Parties in $S \setminus \{p\}$ are Byzantine that act like in r.
- Parties in P are Byzantine. They do not send messages to p, but other than that act as honest parties.
- All other parties, including p, are honest.

First, note that the number of Byzantine parties in r' is $|S| - 1 + |P| \leq f - 1 + t/2 \leq t$. Also, since p acts in r as an honest party that does not receive messages, and all Byzantine parties in r' act towards honest parties in r' $(\Pi \setminus (S \cup P))$ in exactly the same way as they do in r, then honest parties in r' cannot distinguish between r and r'. Thus, since they do not send messages to p in r they do not send in r' as well. Therefore, p does not get any message in r'. ◀

The next Lemma is proven by showing that honest parties that do not get messages cannot safely decide. Not that the case of $f > t/2$ is not required to conclude Theorem 2 since in this case $o(ft) = o(t^2)$.

▶ **Lemma 7.** *For any $1 \leq f \leq \lfloor t/2 \rfloor$, there is no optimistic Byzantine agreement algorithm which synchronous communication cost with f failures is $o(ft)$.*

Proof. Assume by a way of contradiction such protocol A which synchronous communication cost with f failures is $o(ft)$ for some $1 \leq f \leq \lfloor t/2 \rfloor$. Pick a set of $S_1 \subset \Pi$ of f parties and let V be the set of values that honest parties propose. By Lemma 6, there is a run r_1 of A in which honest parties propose values from V s.t. some honest party $p_1 \in S$ does not get any messages. Now let $S_2 = \{p\} \cup S_1 \setminus \{p_1\}$ s.t. $p \in \Pi \setminus S_1$. By Lemma 6 again, there is a run r_2 of A in which honest parties propose values from V s.t. some honest party $p_2 \neq p_1$ does not get any messages. Since $f \leq \lfloor t/2 \rfloor$, we can repeat the above $2t + 1$ times by each time replacing the honest party in S_i that get no messages with a party not in $S_i \cup \{p_1, p_2, \ldots, p_i\}$. Thus, we get that for every possible set of inputs V (values proposed by honest parties) there is a set T of $2t + 1$ parties s.t. for every party $p \in T$ there is a run of A in which honest

parties propose values from V, p is honest, and p does not get any messages. In particular, there exist such set T_0 for the case in which all honest party input 0 and a set T_1 for the case in which all honest parties input 1. Since $|T_0| = |T_1| = 2t + 1$, there is a party $p \in T_1 \cap T_2$. Therefore, by the Termination and Binary validity properties, there is a run r in which p does not get any messages and decides 0 and a run r' in which p does not any messages and decides 1. However, since r and r' are indistinguishable to p we get a contradiction. ◀

The following Theorem follows directly from Lemma 7 and Claim 5.

▶ **Theorem 2** (restated). *Any synchronous deterministic Byzantine agreement protocol has a communication cost of* $\Omega(ft + t)$.

4 Asymptotically optimal optimistic Byzantine Agreement

Our optimistic Byzantine agreement protocol safely combines synchronous and asynchronous protocols. Our synchronous protocol, which is interesting on its own, matches the lower bound proven in Theorem 2. That is, its communication complexity is $O(ft + t)$. The asynchronous protocol we use has a worst-case optimal quadratic communication complexity. For ease of exposition, we construct our protocol in steps. First, in Section 4.1, we present the local state each party maintains, define the *leader-based view (LBV)* [37] building block, which is used by both protocols, and present an implementation with $O(n)$ communication complexity. Then, in Section 4.2, we describe our synchronous protocol, and in Section 4.3 we use the LBV building block to reconstruct VABA [1] - an asynchronous Byzantine agreement protocol with expected $O(n^2)$ communication cost and $O(1)$ running time. Finally, in section 4.4, we safely combine both protocols to prove the following:

▶ **Theorem 4** (restated). *There is an authenticated Byzantine agreement protocol with* $O(ft + t)$ *communication complexity in synchronous runs and expected* $O(t^2)$ *communication complexity in all other runs.*

The correctness proof and communication analysis of the protocol appear in the fullpaper [36].

4.1 General structure

The protocol uses many instances of the LBV building block, each of which is parametrized with a sequence number and a leader. We denote an LBV instance that is parametrized with sequence number sq and a leader p_l as $LBV(sq, p_l)$. Each party in the protocol maintains a local state, which is used by all LBVs and is updated according to their returned values. Section 4.1.1 presents the local state and Section 4.1.2 describes a linear communication LBV implementation. Section 4.1.3 discusses the properties guaranteed by a sequential composition of several LBV instances.

4.1.1 Local state

The local state each party maintains is presented in Algorithm 1. For every possible sequence number sq, $LEADER[sq]$ stores the party that is chosen (a priori or in retrospect) to be the leader associated with sq. The $COMMIT$ variable is a tuple that consists of a value val, a sequence number sq s.t. val was committed in LBV(sq,$LEADERS$[sq]), and a threshold signature that is used as a proof of it. The $VALUE$ variable contains a safe value to propose and the KEY variable is used as proof that $VALUE$ is indeed safe. KEY contains a sequence number sq and a threshold signature that proves that no value other than $VALUE$ could

be committed in LBV(sq,$LEADERS$[sq]). The $LOCK$ variable stores a sequence number sq, which is used to determine what keys are up-to-date and what are obsolete – a key is up-to-date if it contains a sequence number that is greater than or equal to $LOCK$.

Algorithm 1 Local state initialization.

$LOCK \in \mathbb{N} \cup \{\perp\}$, initially $\perp$
$KEY \in (\mathbb{N} \times \{0,1\}^*) \cup \{\perp\}$ with selectors sq and $proof$, initially $\perp$
$VALUE \in \mathbb{V} \cup \{\perp\}$, initially $\perp$
$COMMIT \in (\mathbb{V} \times \mathbb{N} \times \{0,1\}^*) \cup \{\perp\}$ with selectors val, sq and $proof$, initially $\perp$
for every $sq \in \mathbb{N}$, $LEADER[sq] \in \Pi \cup \{\perp\}$, initially $\perp$

4.1.2 Linear leader-based view

For space limitation, detailed pseudocode of the linear implementation of the LBV building block is given in the fullpaper [36]. An illustration appears in figure 1. The LBV building block supports an API to *start the view* and *wedge the view*. Upon a $\mathsf{startView}(\langle \mathsf{sq}, \mathsf{p_l} \rangle)$ invocation, the invoking party starts processing messages associated with LBV(sq,p_l). When the leader p_l invokes $\mathsf{startView}(\langle \mathsf{sq}, \mathsf{p_l} \rangle)$ it initiates 3 steps of leader-to-all and all-to-leader communication, named *PreKeyStep*, *KeyStep*, and *LockStep*. In each step, the leader sends its $VALUE$ together with a threshold signature that proves the safety of the value for the current step and then waits to collect $n - t$ valid replies. A party that gets a message from the leader, validates that the received value and proof are valid for the current step, then produces its signature share on a message that contains the value and the step's name, and sends the share back to the leader. When the leader gets $n - t$ valid shares, it combines them into a threshold signature and continues to the next step. After successfully generating the threshold signature at the end of the third step (*LockStep*), the leader has a commit certificate which he sends together with its $VALUE$ to all parties.

In addition to validating and share-signing messages, parties also store the values and proofs they receive. The **keyProof** and **lockProof** variables store tuples consisting of the values and the threshold signatures received from the leader in the *KeyStep*, and *LockStep* steps, respectively. The **commitProof** variable stores the received value and the commit certificate. When a party receives a valid commit certificate from the leader it returns.

As for the validation of the leader's messages, parties distinguish the *PreKeyStep* message from the rest. For *KeyStep*, *LockStep* and commit certificate messages, parties simply check that the attached proof is a valid threshold signature on the leader's value and the previous step name. The *PreKeyStep* message, however, is used by the Agreement protocols to safely compose many LBV instances. We describe this mechanism in more details below, but to develop some intuition let us first present the properties guaranteed by a single LBV instance:

- Commit causality: If a party gets a valid commit certificate, then at least $t + 1$ honest parties previously got a valid **lockProof**.
- Lock causality: If a party gets a valid **lockProof**, then at least $t + 1$ honest parties previously got a valid **keyProof**.
- Safety: All valid **keyProof**, **lockProof**, and commit certificates obtained in the same LBV have the same value.

The validation of the *PreKeyMessage* in *PreKeyStep* makes sure that the leader's value satisfies the safety properties of the Byzantine agreement protocol that sequentially composes and operates several LBVs. The *PreKeyMessage* contains the leader's $VALUE$ and KEY, where KEY stores the last (non-empty) **keyProof** returned by a previous LBV instance

together with the LBV's sequence number. When a party gets a *PreKeyMessage* it first validates, by checking the key's sequence number *sq*, that the attached key was obtained in an LBV instance that does not precede the one the party is locked on (the sequence number that is stored in the party's *LOCK* variable). Then, the party checks that the threshold signature in the key (1) was generated at the end of the *PreKeyStep* step (it is a valid **keyProof**) in LBV(sq,*LEADER[sk]*); and (2) it is a valid signature on a message that contains the leader's *VALUE*. Note that if the party is not locked ($LOCK = \perp$) then a key is not required.

Upon a wedgeView(sq, p_l) invocation, the invoking party stops participating in LBV(sq,p_l) and returns its current **keyProof**, **lockProof**, and **commitProof** values. These values are used by both synchronous and asynchronous protocols, which are built on top of LBV instances, to update the *LOCK*, *KEY*, *VALUE*, and *COMMIT* variables in parties' local states. Stopping participating in LBV(sq,p_l) upon a wedgeView(sq, p_l) invocation guarantees that the the LBVs' causality guarantees are propagated the *KEY*, *LOCK*, and *COMMIT* variables in parties local states.

Communication complexity. Note that the number of messages sent among honest parties in an LBV instance is $O(n) = O(t)$. In addition, since signatures are not accumulated – leaders use threshold signatures – each message contains a constant number of words, and thus the total communication cost of an LBV instance is $O(t)$ words.

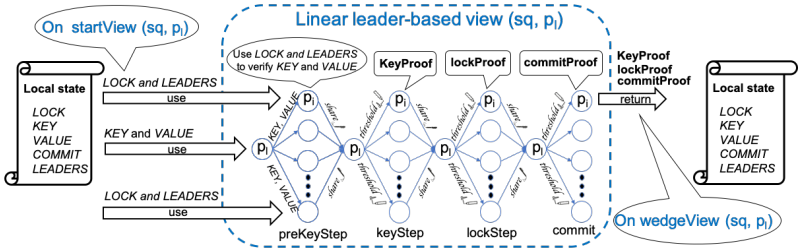


Figure 1 A linear communication LBV illustration. The local state is used by and updated after each instance. The **keyProof**, **lockProof**, and **commitProof** are returned when a commit message is received from the leader or *wedgeView* is invoked.

4.1.3 Sequential composition of LBVs

As mentioned above, our optimistic Byzantine agreement protocol is built on top of the LBV building blocks. The synchronous and the asynchronous parts of the protocol use different approaches, but they both sequentially compose LBVs - the synchronous part of the protocol determines the composition in advance, whereas the asynchronous part chooses what instances are part of the composition in retrospect.

In a nutshell, a sequential composition of LBVs operates as follows: parties start an LBV instance by invoking startView and at some later time (depends on the approach) invoke wedgeView and update their local states with the returned values. Then, they exchange messages to propagate information (e.g., up-to-date keys or commit certificates), update their local states again and start the next LBV (via startView invocation). We claim that an agreement protocol that sequentially composes LBV instances and maintains the local state in Algorithm 1 has the following properties:

- Agreement: all commit certificates in all LBV instances have the same value.
- Conditional progress: for every LBV instance, if the leader is honest, all honest parties invoke *startView*, and all messages among honest parties are delivered before some honest party invokes *wedgeView*, then all honest parties get a commit certificate.

Intuitively, by the LBV's commit causality property, if some party returns a valid commit certificate (**commitProof**) with a value v in some LBV(sq,p_i), then at least $t+1$ honest parties return a valid **lockProof** and thus lock on sq ($LOCK \leftarrow sq$). Therefore, since the leader of the next LBV needs the cooperation of $n-t$ parties to generate threshold signatures, its *PreKeyStep* message must include a valid **keyProof** that was obtained in LBV(sq,p_i). By the LBV's safety property, this **keyProof** includes the value v and thus v is the only value the leader can propose. The agreement property follows by induction.

As for conditional progress, we have to make sure that honest leaders are able to drive progress. Thus, we must ensure that all honest leaders have the most up-to-date keys. By the lock causality property, if some party gets a valid **lockProof** in some LBV, then at least $t+1$ honest parties get a valid **keyProof** in this LBV and thus are able to unlock all honest parties in the next LBV. Therefore, leaders can get the up-to-date key by querying a quorum of $n-t$ parties.

From the above, any Byzantine agreement protocol that sequentially composes LBVs satisfies Agreement. The challenge, which we address in the rest of this section, is how to sequentially compose LBVs in a way that satisfies Termination with asymptotically optimal communication complexity under all network conditions and failure scenarios.

4.2 Adaptive to failures synchronous protocol

Algorithm 2 Adaptive synchronous protocol: Procedure for a party p_i.

```
1:  upon Synch-propose(v_i) do
2:      VALUE ← v_i
3:      tryOptimistic()

4:  procedure TRYOPTIMISTIC()
5:      trySynchrony(1, p_1, 7Δ)
6:      for j ← 2 to n do
7:          if i ≠ j then
8:              trySynchrony(j, p_j, 9Δ)
9:          else if COMMIT = ⊥ then
10:             send "KEYREQUEST" to all parties
11:             wait for 2Δ time
12:             trySynchrony(j, p_j, 7Δ)

13: procedure TRYSYNCHRONY(sq, leader, T)
14:     invoke startView(sq, leader)                    ▷ non-blocking invocation
15:     wait for T time
16:     ⟨keyProof, lockProof, commitProof⟩ ← wedgeView(sq, leader)
17:     updateState(sq, leader, keyProof, lockProof, commitProof)

18: upon receiving "KEYREQUEST" from party p_k for the first time do
19:     send "KEYREPLY, KEY, VALUE" to party p_k

20: upon receiving "KEYREPLY, key, value" do
21:     check&updateKey(key, value)
```

In this section, we describe a synchronous Byzantine agreement protocol with an asymptotically optimal adaptive communication cost that matches the lower bound in Theorem 2. Namely, we prove the following Theorem:

▶ **Theorem 3** (restated). *There is a deterministic synchronous authenticated Byzantine agreement protocol with $O(ft + t)$ communication complexity.*

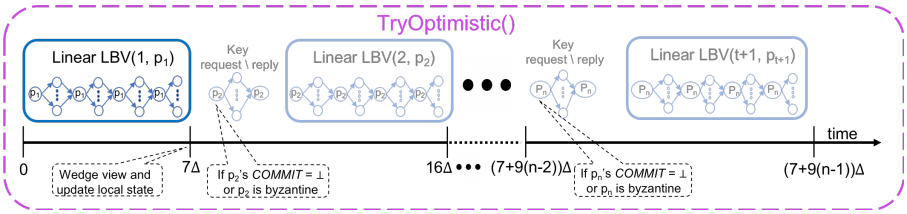


Figure 2 Illustration of the adaptive synchronous protocol. Shaded LBVs are not executed if their leaders have previously decided.

A detailed pseudocode is given in Algorithms 2 and 3, and an illustration appears in Figure 2. The protocol sequentially composes n pre-defined LBV instances, each with a different leader, and parties decide v whenever they get a commit certificate with v in one of them. To avoid the costly view-change mechanism that is usually unavoidable in leader-based protocols, parties exploit synchrony to coordinate their actions. That is, all the startView and wedgeView invocation times are predefined, e.g., the first LBV starts at time 0 and is wedged at time 7Δ simultaneously by all honest parties. In addition, to make sure honest leaders can drive progress, each leader (except the first) learns the up-to-date key, before invoking startView, by querying all parties and waiting for a quorum of $n - t$ parties to reply.

Algorithm 3 Auxiliary procedures to update local state.

```
1:  procedure updateState(sq, leader, keyProof, lockProof, commitProof)
2:      LEADERS[sq] ← leader
3:      if keyProof ≠ ⊥ then
4:          KEY ← ⟨sq, keyProof.proof⟩
5:          VALUE ← keyProof.val
6:      if lockProof ≠ ⊥ then
7:          LOCK ← sq
8:      if commitProof ≠ ⊥ then
9:          COMMIT ← ⟨commitProof.val, sq, commitProof.proof⟩
10:         decide COMMIT.val

11: procedure check&updateKey(key, value)
12:     if (KEY = ⊥ ∨ key.sq > KEY.sq) then
13:         if threshold-validate(⟨preKeyStep, key.sq,
14:                 LEADER[key.sq], value⟩, key.proof) then
15:             KEY ← key
16:             VALUE ← value

17: procedure check&updateCommit(commit)
18:     if COMMIT = ⊥ then
19:         if threshold-validate(⟨lockStep, commit.sq,
20:           LEADER[commit.sq], commit.val⟩, commit.proof) then
21:             COMMIT ← commit
22:             decide COMMIT.val
```

Composing n LBV instances may lead in the worst case to $O(t^2)$ communication complexity – $O(t)$ for every LBV instance. Therefore, to achieve the optimal adaptive complexity, honest leaders in our protocol participate (learn the up-to-date key and invoke startView) only in case they have not yet decided. (Note that the communication cost of an LBV instance in which the leader does not invoke startView is 0 because other parties only reply to the

leader's messages.) For example, if the leader of the second LBV instance is honest and has committed a value in the first instance (its $COMMIT \neq \perp$ at time 7Δ), then no message is sent among honest parties between time 7Δ and time 16Δ.

Termination and communication complexity. A naive approach to guarantee termination and avoid an infinite number of LBV instances in a leader based Byzantine agreement protocols is to perform a costly communication phase after each LBV instance. One common approach is to reliably broadcast commit certificates before halting, while a complementary one is to halt unless receiving a quorum of complaints from parties that did not decide. In both cases, the communication cost is $O(t^2)$ even in runs with one failure.

The key idea of our synchronous protocol is to exploit synchrony in order to allow honest parties to learn the decision value and at the same time help others in a small number of messages. Instead of complaining (together) after every unsuccessful LBV instance, each party has its own pre-defined time to "complain", in which it learns the up-to-date key and value and helps others decide via the LBV instance in which it acts as the leader.

By the conditional progress property and the synchrony assumption, all honest parties get a commit certificate in LBV instances with honest leaders. Therefore, the termination property is guaranteed since every honest party has its own pre-defined LBV instance, which it invokes only in case it has not yet decided. As for the protocol's total communication cost, recall that the LBV's communication cost is $O(t)$ in the worst case and 0 in case its leader already decided and thus does not participate. In addition, since all honest parties get a commit certificate in the first LBV instance with an honest leader, we get that the message cost of all later LBV instances with honest leaders is 0. Therefore, the total communication cost of the protocol is $O(ft + t)$ – at most f LBVs with Byzantine leaders and 1 LBV with an honest one.

4.3 Asynchronous fallback

In this section, we use the LBV building block to reconstruct VABA [1]. Note that achieving an optimal asynchronous protocol is not a contribution of this paper but reconstructing the VABA protocol with our LBV building block allows us to safely combine it with our adaptive synchronous protocol to achieve an optimal optimistic one. In addition, we also improve the protocol of VABA in the following ways: first, parties in VABA [1] never halt, meaning that even though they decide in expectation in a constant number of rounds, they operate an unbounded number of them. We fix it by adding an auxiliary primitive, we call *help&tryHalting* in between two consecutive waves. Second, VABA guarantees probabilistic termination in all runs, whereas our version also guarantees standard termination in eventually synchronous runs. For space limitation, the details are given in Appendix B.

4.4 Optimal optimistic protocol: combine the pieces

Algorithm 4 Optimistic byzantine agreement: protocol for a party p_i.

```
1: upon Optimistic-propose(v_i) do
2:     VALUE ← v_i
3:     tryOptimistic()
4:     help&tryHalting(n)                    ▷ Blocking invocation
5:     fallback(n)
```

At a high level, parties first optimistically try the synchronous protocol (of section 4.2), then invoke *help&tryHalting* and continue to the asynchronous fallback (of section 4.3) in case a decision has not been reached. Pseudocode is given in Algorithm 4 and an illustration appears in Figure 3. The parameters passed in Algorithm 4 synchronize the LBV sequence numbers across the different parts of the protocol.

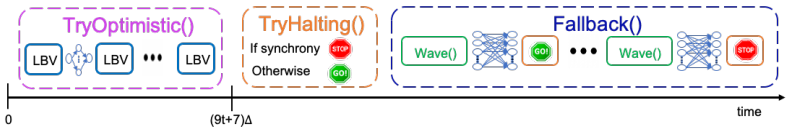


Figure 3 Illustration of the optimistic protocol. Both parts form a sequential composition of LBV instances.

One of the biggest challenges in designing an agreement protocol as a combination of other protocols is to make sure safety is preserved across them. Meaning that parties must never decide differently even if they decide in different parts of the protocol. In our protocol, however, this is inherently not a concern. Since both parts use LBV as a building block, we get safety for free. That is, if we look at an execution of our protocol in retrospect, i.e, ignore all LBVs that were not elected in the asynchronous part. Then the LBV instances in the synchronous part together with the elected ones in the asynchronous part form a sequential composition, which satisfies the Agreement property.

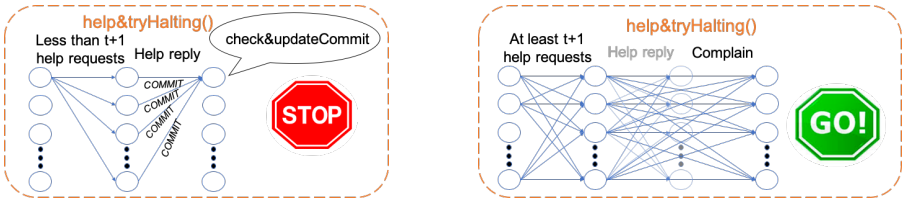
On the other hand, satisfying termination without sacrificing optimal adaptive complexity is a non-trivial challenge. Parties start the protocol by optimistically trying the synchronous part, but unfortunately, at the end of the synchronous part they cannot distinguish between the case in which the communication was indeed synchronous and all honest parties decided and the case in which some honest parties did not decide due to asynchrony. Moreover, honest parties cannot distinguish between honest parties that did not decide and thus wish to continue to the asynchronous fallback part and Byzantine parties that want to move to the fallback part to increase the communication cost.

To this end, we implement the *help&tryHalting* procedure, which stops honest parties from moving to the fallback part in synchronous runs. The communication cost of *help&tryHalting* is $O(ft)$. The idea is to help parties learn the decision value and move to the fallback part only when the number of help request indicates that the run is asynchronous.

The pseudocode of *help&tryHalting* is given in Appendix D and an illustration appears in Figure 4. Each honest party that has not yet decided sends a share signed HELPREQUEST to all other parties. When an honest party gets an HELPREQUEST, it replies with its *COMMIT* value. But if it gets $t + 1$ HELPREQUEST messages, the party combines the shares to a threshold signature and sends it in a COMPLAIN message to all. When an honest party gets a COMPLAIN message for the first time, it echos the message to all parties and continues to the fallback part. A termination intuition and complexity analysis of our full protocol are given in Appendix C.

5 Discussion and Future Directions

In this paper, we propose a new approach to design agreement algorithms for communication efficient SMR systems. Instead of designing deterministic protocols for the eventually synchronous model, which we prove cannot guarantee bounded communication cost before GST, we propose to design protocols that are optimized for the synchronous case but also have a randomized fallback to deal with asynchrony. Traditionally, most SMR solutions



(a) A few HELPREQUEST messages – help and halt.

(b) Too much HELPREQUEST messages – the run is asynchronous, move to the fallback part.

Figure 4 An illustration of the *help&tryHalting* procedure.

avoid randomized asynchronous protocols due to their high communication cost. We, in contrast, argue that this communication cost is reasonable given that the alternative is an unbounded communication cost during the wait for eventual synchrony.

We present the first authenticated optimistic protocol with $O(ft + t)$ communication complexity in synchronous runs and $O(t^2)$, in expectation, in non-synchronous runs. To strengthen our result, we prove that no deterministic protocol (even if equipped with perfect cryptographic schemes) can do better in synchronous runs. As for the asynchronous runs, the lower bound in[1] proves that $O(t^2)$ is optimal in the worst case of $f = t$.

Future work. Note that our synchronous protocol satisfies early decision but not early stopping. That is, all honest parties decide after $O(f)$ rounds, but they terminate after $O(t)$. Therefore, a natural question to ask is whether exist an early stooping synchronous Byzantine agreement protocol with an optimal adaptive communication cost. In addition, it may be possible to improve our protocol's complexity even further. In particular, the lower bound on communication cost in synchronous runs applies only to deterministic algorithms, so it might be possible to circumvent it via randomization [8].

Another interesting future direction is the question of optimal resilience in synchronous networks. Due to the lower bound in [5], the resilience of our protocol is optimal since the resilience in synchronous runs cannot be improved as long as the resilience in asynchronous runs is the optimal $t < n/3$. However, if we consider synchronous networks in which we do not need to worry about asynchronous runs, we know that we can tolerate up to $t < n/2$ failures. The open question is therefore the following: is there a synchronous Byzantine agreement protocol that tolerates up to $t < n/2$ failures with an optimal communication complexity of $O(ft + t)$?

References

1 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *PODC*. ACM, 2019.

2 Marcos Kawazoe Aguilera and Sam Toueg. Randomization and failure detection: A hybrid approach to solve consensus. In *IWDA*, 1996.

3 Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Sok: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 183–198, 2019.

4 Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. In *Theory of Cryptography Conference*, 2020.

5 Erica Blum, Jonathan Katz, and Julian Loss. Synchronous consensus with optimal asynchronous fallback guarantees. In *Theory of Cryptography Conference*. Springer, 2019.

6 Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

7 Francisco Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *International Conference on Parallel Computing Technologies*, 2001.

8 Nicolas Braud-Santoni, Rachid Guerraoui, and Florian Huc. Fast byzantine agreement. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, 2013.

9 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. In *DISC*. Schloss Dagstuhl-Leibniz-Zentrum fur Informatik, 2020.

10 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint*, 2018. `arXiv:1807.04938`.

11 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology*, 2001.

12 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, 1999.

13 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: Sub-quadratic asynchronous byzantine agreement whp. In *DISC 2020*, 2020.

14 Danny Dolev and Rudiger Reischuk. Bounds on information exchange for byzantine agreement. *JACM*, 1985.

15 Danny Dolev, Ruediger Reischuk, and H Raymond Strong. Early stopping in byzantine agreement. *Journal of the ACM (JACM)*, 1990.

16 Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

17 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 1985.

18 Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, 2010.

19 Rachid Guerraoui, Viktor Kuncak, and Giuliano Losa. Speculative linearizability. *ACM Sigplan Notices*, 47(6):55–66, 2012.

20 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. In *DSN 2019*. IEEE, 2019.

21 Idit Keidar and Sergio Rajsbaum. A simple proof of the uniform consensus synchronous lower bound. *Information Processing Letters*, 2003.

22 Valerie King and Jared Saia. Byzantine agreement in expected polynomial time. *Journal of the ACM (JACM)*, 63(2):13, 2016.

23 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, 2007.

24 Kfir Lev-Ari, Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Fairledger: A fair blockchain protocol for financial institutions. In *OPODIS*, 2019.

25 Dahlia Malkhi. *Concurrency: The Works of Leslie Lamport*. Morgan & Claypool, 2019.

26 Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *CCS*, 2019.

27 J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

28 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *CCS 2016*. ACM, 2016.

29 Atsuki Momose and Ling Ren. Optimal communication complexity of byzantine consensus under honest majority. *DISC*, 2020.

30 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. In *CryptoEconSys 2020*, 2019.

31 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. In *DISC*, 2020.

32 Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. *DISC*, 2020.

33 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2), 1980.

34 Victor Shoup. Practical threshold signatures. In *ICTACT*, 2000.

35 Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In *International Symposium on Distributed Computing*, pages 438–450. Springer, 2008.

36 Alexander Spiegelman. In search for an optimal authenticated byzantine agreement. *CoRR*, abs/2002.06993, 2020. `arXiv:2002.06993`.

37 Alexander Spiegelman, Arik Rinberg, and Dahlia Malkhi. Ace: Abstract consensus encapsulation for liveness boosting of state machine replication. In *OPODIS' 2020*, 2020.

38 Maofan Yin, Dahlia Malkhi, MK Reiterand, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *PODC*, 2019.

A Lower Bound For Eventually Synchronous Runs

▶ **Theorem 1** (restated). *There is no eventually synchronous deterministic Byzantine agreement protocol that can tolerate a single failure and guarantee bounded communication cost even in failure-free runs.*

Proof. Assume by a way of contradiction that there are such algorithms. Let A be such an algorithm with the lowest eventually synchronous communication cost with 0 failures, and denote its communication cost by N. Clearly, $N \geq 1$. Let $R_N \subset R_0^{es}$ be the set of all failure-free eventually synchronous runs of A that have communication cost of N. For every run $r \in R_N$ let m_r be the last message that is delivered in r, let t_r be the time at which it is delivered, and let p_r be the party that sends m_r. Now for every $r \in R_N$ consider a run r' that is identical to r up to time t_r except p_r is Byzantine that acts exactly as in r but does not send m_r. Denote by R_{N-1} the set of all such runs and consider two cases:

- There is a run $r' \in R_{N-1}$ in which some message m by an honest party p is sent at some time $t_{r'} > t_r$. Now consider a failure-free run r'' that is identical to run r except the delivery of m_r is delayed to $t_{r'} + 1$. The runs r'' and r' are indistinguishable to all parties that are honest in r' and thus p sends m at time time $t_{r'} > t_r$ in r'' as well. Therefore, the communication cost of r'' is at least $N + 1$. A contradiction to the communication cost of A.
- Otherwise, we can construct an algorithm A' with a better eventually synchronous communication cost with 0 failures than A in the following way: A' operates identically to A in all runs not in R_N and for every run $r \in R_N$ A' operates as A except p_r does not send m_r. A contradiction to the definition of A. ◀

B Fallback Description

On a high level, the idea in VABA [1] that was later generalized in ACE [37] is the following: instead of having a pre-defined leader in every "round" of the protocol as most eventually synchronous protocols have, they let n leaders operate simultaneously and then randomly choose one in retrospect. This mechanism is implemented inside a wave and the agreement protocol operates in a *wave*-by-*wave* manner s.t. parties exchange their local states between every two conductive waves. To ensure halting, in our version of the protocol, parties also invoke the *help&tryHalting* procedure after each wave. See Figure 5 for an illustration. A full detailed pseudocode of our fallback protocol can be found in the fullpaper [36].

Wave-by-wave approach. To implement the wave mechanism we use our LBV and two auxiliary primitives: Leader-election and Barrier-synchronization. At the beginning of every wave, parties invoke, via startView, n different LBV instances, each with a different leader.

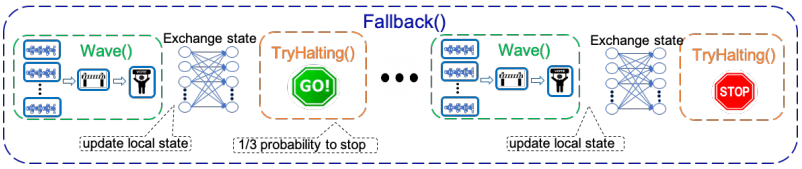


Figure 5 Asynchronous fallback. Usig linear LBV to reconstruct the VABA [1] protocol.

Then, parties are blocked in the Barrier-synchronization primitive until at least $n - 2t$ LBV instances *complete*. (An LBV completes when $t + 1$ honest parties get a commit certificate.) Finally, parties use the Leader-election primitive to elect a unique LBV instance, wedge it (via wedgeView), and ignore the rest. With a probability of 1/3 parties choose a completed LBV, which guarantees that after the state exchange phase all honest parties get a commit certificate, decide, and halt in the *help&tryHalting* procedure. Otherwise, parties update their local state and continue to the next wave. An illustration appears in figure 6.

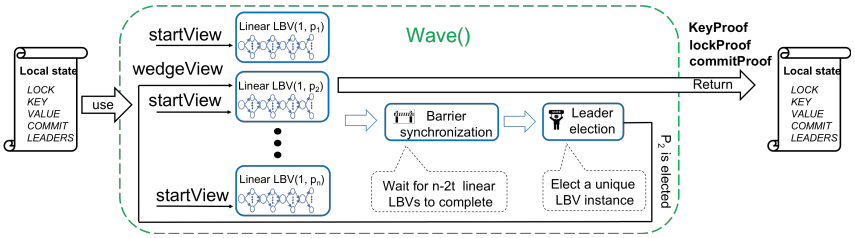


Figure 6 An illustration of a single wave. The returned **keyProof**, **lockProof**, and **commitProof** are taken from the elected LBV.

Since every wave has a probability of 1/3 to choose a completed LBV instance, the protocol guarantees probabilistic termination – in expectation, all honest parties decide after 3 waves. To also satisfy standard termination in eventually synchronous runs, we "try synchrony" after each unsuccessful wave. Between every two conjunctive waves parties deterministically try to commit a value in a pre-defined LBV instance. The preceding *help&tryHalting* procedure guarantees that after GST all honest parties invoke startView in the pre-defined LBV instance with at most 1Δ from each other and thus setting a timeout to 8Δ is enough for an honest leader to drive progress. Description of the Barrier-synchronization and Leader-election primitives can be found in [37].

C Protocol Termination Intuition And Complexity Analysis

Termination. A formal proof of Safety and Liveness is given in the fullpaper [36]. Here we provide some intuition. Consider two cases. First, the parties move to the fallback part, in which case (standard) termination is guaranteed in eventually synchronous runs and probabilistic termination is guaranteed in asynchronous runs. Otherwise, less than $t + 1$ parties send HELPREQUEST in *help&tryHalting*, which implies that at least $t+1$ honest parties decided and had a commit certificate before invoking *help&tryHalting*. Therefore, all honest parties that did not decide before invoking *help&tryHalting* eventually get a HELPREPLY message with a commit certificate and decide as well.

Note that termination does not mean halting. In asynchronous runs, HELPREQUEST messages may be arbitrary delayed and thus parties cannot halt the protocol after deciding in the synchronous part. However, it is well known and straightforward to prove that halting cannot be achieved with $o(t^2)$ communication cost in asynchronous runs, and thus our protocol is optimal in this aspect.

Round complexity. Since in synchronous runs all parties decide at the end of an LBV instances with an honest leader, we get that the round complexity in synchronous runs is $O(f+1)$. Since in asynchronous runs parties may go though n LBV instances without deciding before starting the fallback, we get that the round complexity in asynchronous runs is $O(n+1)$ in expectations.

Communuication complexity. The synchronous (optimistic) part guarantees that if the run is indeed synchronous, then all honest parties decide before invoking *help&tryHalting*. The *help&tryHalting* procedure guarantees that parties continue to the fallback part only if $t+1$ parties send an HELPREQUEST message, which implies that they move only if at least one honest party has not decided in the synchronous part. Therefore, together they guarantee that honest parties never move to the fallback part in synchronous runs.

The communication complexity of the synchronous part is $O(ft+t)$, so to show that the total communication cost of the protocol in synchronous runs is $O(ft+t)$ we need to show that the cost of *help&tryHalting* is $O(ft+t)$ as well. Since in synchronous runs all honest parties decide in the synchronous part, they do not send HELPREQUEST messages, and thus no party can send a valid COMPLAIN message. Each Byzantine party that does send HELPREQUEST messages can cause honest parties to send $O(t)$ replies, which implies a total communication cost of $O(ft)$ in synchronous runs.

As for all other runs, Theorem 1 states that deterministic protocols have an unbounded communication cost in the worst case. Thanks to the randomized fallback, our protocol has a communication cost of $O(t^2)$ in expectation.

D Help&tryHalting Pseudocode

Algorithm 5 Help and try halting: Procedure for a party p_i.

Local variables initialization:
$S_{help} = \{\}$; $HALT \leftarrow true$

1: **procedure** help&tryHalting(sq)
2: **if** $COMMIT = \perp$ **then**
3: $\rho \leftarrow$ *share-sign*$_i(\langle$HELPREQUEST, $sq\rangle)$
4: send "HELPREQUEST, sq, ρ" to all parties
5: **wait** until $HALT = false$

6: **upon receiving** "HELPREPLY, sq, *commit*" **do**
7: check&updateCommit(*commit*)

8: **upon receiving** "HELPREQUEST, sq, ρ" from a party p_j **do**
9: **if** *share-validate*$(\langle$HELPREQUEST, $sq\rangle, p_j, \rho)$ **then**
10: $S_{help} \leftarrow S_{help} \cup \{\rho\}$
11: send "HELPREPLY, sq, *COMMIT*" to p_j
12: **if** $|S_{help}| = t+1$ **then**
13: $\nu \leftarrow$ *threshold-sign*(S_{help})
14: send "COMPLAIN, sq, ν" to all parties

15: **upon receiving** "COMPLAIN, sq, ν" **do**
16: **if** *threshold-validate*$(\langle$HELPREQUST, $sq\rangle, \nu)$ **then**
17: send "COMPLAIN, sq, ν" to all parties
18: $HALT \leftarrow false$