Frugal Byzantine Computing

Marcos K. Aguilera
VMware Research, Palo Alto, CA, USA

Naama Ben-David
VMware Research, Palo Alto, CA, USA

Rachid Guerraoui
EPFL, Lausanne, Switzerland

Dalia Papuc
EPFL, Lausanne, Switzerland

Athanasios Xygkis
EPFL, Lausanne, Switzerland

Igor Zablotchi
MIT, Cambridge, MA, USA

Abstract

Traditional techniques for handling Byzantine failures are expensive: digital signatures are too costly, while using $3f+1$ replicas is uneconomical (f denotes the maximum number of Byzantine processes). We seek algorithms that reduce the number of replicas to $2f+1$ and minimize the number of signatures. While the first goal can be achieved in the message-and-memory model, accomplishing the second goal simultaneously is challenging. We first address this challenge for the problem of broadcasting messages reliably. We study two variants of this problem, Consistent Broadcast and Reliable Broadcast, typically considered very close. Perhaps surprisingly, we establish a separation between them in terms of signatures required. In particular, we show that Consistent Broadcast requires at least 1 signature in some execution, while Reliable Broadcast requires $O(n)$ signatures in some execution. We present matching upper bounds for both primitives within constant factors. We then turn to the problem of consensus and argue that this separation matters for solving consensus with Byzantine failures: we present a practical consensus algorithm that uses Consistent Broadcast as its main communication primitive. This algorithm works for $n = 2f+1$ and avoids signatures in the common case – properties that have not been simultaneously achieved previously. Overall, our work approaches Byzantine computing in a frugal manner and motivates the use of Consistent Broadcast – rather than Reliable Broadcast– as a key primitive for reaching agreement.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms; Theory of computation → Distributed algorithms; Theory of computation → Design and analysis of algorithms

Keywords and phrases Reliable Broadcast, Consistent Broadcast, Consensus, Byzantine Failure, Message-and-memory

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.3

Related Version *Full Version*: `https://arxiv.org/abs/2108.01330` [4]

1 Introduction

Byzantine fault-tolerant computing is notoriously expensive. To tolerate f failures, we typically need $n = 3f + 1$ replica processes. Moreover, the agreement protocols for synchronizing the replicas have a significant latency overhead. Part of the overhead comes from network delays, but digital signatures – often used in Byzantine computing – are even more costly than network delays. For instance, signing a message can be 28 times slower than sending it over a low-latency Infiniband fabric (Appendix A shows the exact measurements).

In this work, we study whether Byzantine computing can be *frugal*, meaning if it can use few processes and few signatures. By Byzantine computing, we mean the classical problems of broadcast and consensus. By frugality, we first mean systems with $n = 2f + 1$ processes, where f is the maximum number of Byzantine processes. Such systems are clearly preferable to systems with $n = 3f + 1$, as they require 33–50% less hardware. However, seminal impossibility results imply that in the standard message-passing model with $n = 2f + 1$ processes, neither consensus nor various forms of broadcast can be solved, even under partial synchrony or randomization [31]. To circumvent the above impossibility results, we consider a message-and-memory (M&M) model, which allows processes to both pass messages and share memory, capturing the latest hardware capabilities of enterprise servers [1, 2]. In this model, it is possible to solve consensus with $n = 2f + 1$ processes and partial synchrony [2].

Frugality for us also means the ability to achieve *low latency*, by minimizing the number of digital signatures used. Mitigating the cost of digital signatures is commonly done by replacing them with more computationally efficient schemes, such as message authentication codes (MACs). For instance, with $n = 3f+1$, the classic PBFT replaces some of its signatures with MACs [21], while Bracha's broadcast algorithm [15] relies exclusively on MACs. As we show, when $n = 2f + 1$, the same signature-saving techniques are no longer applicable.

The two goals – achieving high failure resilience while minimizing the number of signatures – prove challenging when combined. Intuitively, this is because with $n = 2f + 1$ processes, two quorums may intersect only at a Byzantine process; this is not the case with $n = 3f + 1$. Thus, we cannot rely on quorum intersection alone to ensure correctness; we must instead restrict the behavior of Byzantine processes to prevent them from providing inconsistent information to different quorums. Signatures can restrict Byzantine processes from lying, but only if there are enough correct processes to exchange messages and cross-check information. The challenge is to make processes prove that they behave correctly, based on the information they received so far, while using as few signatures as possible.

We focus initially on the problem of broadcasting a message reliably – one of the simplest and most widely used primitives in distributed computing. Here, a designated sender process s would like to send a message to other processes, such that all correct processes deliver the same message. The difficulty is that a Byzantine sender may try to fool correct processes to deliver different messages. Both broadcast variants, Consistent and Reliable Broadcast, ensure that (1) if the sender is correct, then all correct processes deliver its message, and (2) any two correct processes that deliver a message must deliver the *same* message. Reliable Broadcast ensures an additional property: if any correct process delivers a message, then all correct processes deliver that message.

Perhaps surprisingly, in the M&M model we show a large separation between the two broadcasts in terms of the number of signatures (by correct processes) they require. We introduce a special form of indistinguishability argument for $n = 2f + 1$ processes that uses signatures and shared memory in an elaborate way. With it, we prove lower bounds for deterministic algorithms. For Consistent Broadcast, we prove that any solution requires one correct process to sign in some execution, and provide an algorithm that matches this bound. In contrast, for Reliable Broadcast, we show that any solution requires at least $n - f - 2$ correct processes to sign in some execution. We provide an algorithm for Reliable Broadcast based on our Consistent Broadcast algorithm which follows the well-known Init-Echo-Ready pattern [15] and uses up to $n + 1$ signatures, matching the lower bound within a factor of 2.

To lower the impact of signatures on the latency of our broadcast algorithms, we introduce the technique of background signatures. Given the impossibility of completely eliminating signatures, we design our protocols such that signatures are not used in well-behaved

executions, i.e., when processes are correct and participate within some timeout. In other words, both broadcast algorithms generate signatures in the background and also incorporate a fast path where signatures are not used.

We next show how to use our Consistent Broadcast algorithm to improve consensus algorithms. The algorithm is based on PBFT [20], and maintains *views* in which one process is the *primary*. Within a view, agreement can be reached by simply having the primary consistent-broadcast a value, and each replicator respond with a consistent broadcast. When changing views, a total of $O(n^2)$ calls to Consistent Broadcast may be issued. The construction within a view is similar to our Reliable Broadcast algorithm. Interestingly, replacing this part with the Reliable Broadcast abstraction does *not* yield a correct algorithm; the stronger abstraction hides information that an implementation based on Consistent Broadcast can leverage. For the correctness of our algorithm, we rely on a technique called *history validation* and on *cross-validating* the view-change message. Our consensus algorithm has four features: (1) it works for $n = 2f + 1$ processes, (2) it issues no signatures on the fast path, (3) it issues $O(n^2)$ signatures on a view-change and (4) it issues $O(n)$ background signatures within a view. As far as we know, no other algorithm achieves all these features simultaneously. This result provides a strong motivation for the use of Consistent Broadcast – rather than Reliable Broadcast – as a first-class primitive in the design of agreement algorithms.

To summarize, we quantify the impossibility of avoiding signatures by proving lower bounds on the number of signatures required to solve the two variants of the broadcast problem – *Consistent* and *Reliable Broadcast* – and provide algorithms that match our lower bounds. Also, we construct a practical consensus algorithm using the Consistent Broadcast primitive. In this work, we consider the message-and-memory model [1, 2], but our results also apply to the pure shared memory model: our algorithms do not require messages so they work under shared memory, while our lower bounds apply *a fortiori* to shared memory.

2 Related Work

Message-and-memory models. We adopt a message-and-memory (M&M) model, which is a generalization of both message-passing and shared-memory. M&M is motivated by enterprise servers with the latest hardware capabilities – such as RDMA, RoCE, Gen-Z, and soon CXL – which allow machines to *both* pass messages and share memory. M&M was introduced by Aguilera et al. in [1], and subsequently studied in several other works [2, 6, 33, 47]. Most of these works did not study Byzantine fault tolerance, but focused on crash-tolerant constructions when memory is shared only by subsets of processes [1, 6, 33, 47]. In [2], Aguilera et al. consider crash- and Byzantine- fault tolerance, as well as bounds on communication rounds on the fast path for a variant of the M&M model with dynamic access permissions and memory failures. However, they did not study any complexity bounds off the fast path, and in particular did not consider the number of signatures such algorithms require.

Byzantine Fault Tolerance. Lamport, Shostak and Pease [40, 46] show that Byzantine agreement can be solved in synchronous message-passing systems iff $n \geq 3f + 1$. In asynchronous systems subject to failures, consensus cannot be solved [32]. However, this result is circumvented by making additional assumptions for liveness, such as randomization [10, 45] or partial synchrony [23, 31]. Even with signatures, asynchronous Byzantine agreement can be solved in message-passing systems only if $n \geq 3f + 1$ [17]. Dolev and Reischuk [30] prove a lower bound of $n(f + 1)/4$ signatures for Byzantine agreement, assuming that every message carries at least the signature of its sender.

Byzantine Broadcast. In the message-passing model, both Consistent and Reliable Broadcast require $n \geq 3f + 1$ processes, unless (1) the system is synchronous and (2) digital signatures are available [17, 29, 50]. Consistent Broadcast is sometimes called Crusader Agreement [29]. The Consistent Broadcast abstraction was used implicitly in early papers on Byzantine broadcast [16, 52], but its name was coined later by Cachin et al. in [19]. The name "consistent broadcast" may also refer to a similar primitive used in synchronous systems [42, 50]. Our Reliable Broadcast algorithm shares Bracha's Init-Echo-Ready structure [15] with other broadcast algorithms [17, 48, 50], but is the first algorithm to use this structure in shared memory to achieve Reliable Broadcast with $n = 2f + 1$ processes.

BFT with stronger communication primitives. Despite the known fault tolerance bounds for asynchronous Byzantine Failure Tolerance (BFT), Byzantine consensus can be solved in asynchronous systems with $2f + 1$ processes if stronger communication mechanisms are assumed. Some prior work solves Byzantine consensus with $2f + 1$ processes using specialized trusted components that Byzantine processes cannot control [24, 25, 26, 27, 34, 53]. These trusted components can be seen as providing a broadcast primitive for communication. These works assume the existence of such primitives as black boxes, and do not study the cost of implementing them using weaker hardware guarantees, as we do in this paper. We achieve the same Byzantine fault-tolerance by using the shared memory to prevent the adversary from partitioning correct processes: once a correct process writes to a register, the adversary cannot prevent another correct process from seeing the written value.

It has been shown that shared memory primitives can be useful in providing BFT if they have *access control lists* or *policies* that dictate the allowable access patterns in an execution [2, 5, 11, 13, 43]. Alon et al. [5] provide tight bounds for the number of strong shared-memory objects needed to solve consensus with optimal resilience. They do not, however, study the number of signatures required.

Early termination. The idea of having a *fast path* that allows early termination in well-behaved executions is not a new one, and has appeared in work on both message-passing [2, 3, 7, 28, 35, 36, 39] and shared-memory [8, 51] systems. Most of these works measure the fast path in terms of the number of message delays (or network rounds trips) they require, but some also consider the number of signatures [7]. In this paper, we show that a signature-free fast path does not prevent an algorithm from having an optimal number of overall signatures.

3 Model and Preliminaries

We consider an asynchronous message-and-memory model, which allows processes to use both message-passing and shared-memory [1]. The system has n processes $\Pi = \{p_1, \ldots, p_n\}$ and a shared *memory* M. Throughout the paper, the term memory refers to M, not to the local state of processes. We sometimes augment the system with eventual synchrony (§3.2).

Communication. The memory consists of single-writer multi-reader (SWMR) read/write atomic registers. Each process can read all registers, and has access to an unlimited supply of registers it can write. If a process p can write to a register r, we say that p *owns* r. This model is a special case of access control lists (ACLs) [43], and of dynamically permissioned memory [2]. Additionally, every pair of processes p and q can send messages to each other over links that satisfy the *integrity* and *no-loss* properties. Integrity requires that a message m from p be received by q at most once and only if m was previously sent by p to q. No-loss requires that a message m sent from p to q be eventually received by q.

Signatures. Our algorithms assume digital signatures: each process can *sign* and *verify* signatures. A process p may sign a value v, producing $\sigma_{p,v}$; when unambiguous, we drop the subscripts. Given v and $\sigma_{p,v}$, a process can verify whether $\sigma_{p,v}$ is a valid signature of v by p.

Failures. Up to f processes may fail by becoming Byzantine, where $n = 2f + 1$. Such a process can deviate arbitrarily from the algorithm, but cannot write on a register that is not its own, and cannot forge the signature of a correct process. As usual, Byzantine processes can collude, e.g., by using side-channels to communicate. The memory M does not fail; such a reliable memory is implementable from a collection of fail-prone memories [2]. We assume that these individual memories may only fail by crashing.

3.1 Broadcast

We consider two broadcast variants: Consistent Broadcast [18, 19] and Reliable Broadcast [14, 18]. In both variants, broadcast is defined in terms of two primitives: *broadcast*(m) and *deliver*(m). A designated *sender* process s is the only one that can invoke *broadcast*. When s invokes *broadcast*(m) we say that s *broadcasts* m. When a process p invokes *deliver*(m), we say that p *delivers* m.

▶ **Definition 3.1.** Consistent Broadcast *has the following properties:*

Validity *If a correct process s broadcasts m, then every correct process eventually delivers m.*

No duplication *Every correct process delivers at most one message.*

Consistency *If p and p' are correct processes, p delivers m, and p' delivers m', then $m{=}m'$.*

Integrity *If some correct process delivers m and s is correct, then s previously broadcast m.*

▶ **Definition 3.2.** Reliable Broadcast *has the following properties:*

Validity, No duplication, Consistency, Integrity *Same properties as in Definition 3.1.*

Totality *If some correct process delivers m, then every correct process eventually delivers a message.*

We remark that both broadcast variants behave the same way when the sender is correct and broadcasts m. However, when the sender is faulty Consistent Broadcast has no delivery guarantees for correct processes, i.e., some correct processes may deliver m, others may not. In contrast, Reliable Broadcast forces every correct process to eventually deliver m as soon as one correct process delivers m.

3.2 Consensus

▶ **Definition 3.3.** Weak Byzantine agreement [37] *has the following properties:*

Agreement *If correct processes i and j decide val and val', respectively, then $val = val'$.*

Weak validity *If all processes are correct and some process decides val, then val is the input of some process.*

Integrity *No correct process decides twice.*

Termination *Eventually every correct process decides.*

Our consensus algorithm (§6) satisfies agreement, validity, and integrity under asynchrony, but requires eventual synchrony for termination. That is, we assume that for each execution there exists a *Global Stabilization Time (GST)*, unknown to the processes, such that from GST onwards there is a known bound Δ on communication and processing delays.

4 Lower Bounds on Broadcast Algorithms

We show lower bounds on the number of signatures required to solve Consistent and Reliable Broadcast with $n = 2f+1$ processes in our model. We focus on signatures by correct processes because Byzantine processes can behave arbitrarily (including signing in any execution).

4.1 High-Level Approach

Broadly, we use indistinguishability arguments that create executions E_v and E_w that deliver different messages v and w; then we create a composite execution E where a correct process cannot distinguish E from E_v, while another correct process cannot distinguish E from E_w, so they deliver different values, a contradiction. Such arguments are common in message-passing system, where the adversary can prevent communication by delaying messages between correct processes. However, it is not obvious how to construct this argument in shared memory, as the adversary cannot prevent communication via the shared memory, especially when using single-writer registers that cannot be overwritten by the adversary. Specifically, if correct processes write their values and read all registers, then for any two correct processes, at least one sees the value written by the other [9]. So, when creating execution E in which, say E_v occurs first, processes executing E_w will know that others executed E_v beforehand.

We handle this complication in two ways, depending on whether the sender signs its broadcast message. If the sender does not sign, we argue that processes executing E_w cannot tell whether E_v was executed by correct or Byzantine processes, and must therefore still output their original value w. This is the approach in the lower bound proof for Consistent Broadcast (Lemma 4.1).

However, once a signature is produced, processes can save it in their memory to prove to others that they observed a valid signature. Thus, if the sender signs its value, then processes executing E_w cannot be easily fooled; if they see two different values signed by the sender, then the sender is provably faulty, and correct processes can choose a different output. So, we need another way to get indistinguishable executions. We rely on a *correct bystander* process. We make a correct process b in E sleep until all other correct processes decide. Then b wakes up and observes that E is a composition of E_v and E_w. While b can recognize that E_v or E_w was executed by Byzantine processes, it cannot distinguish which one. So, b cannot reliably output the same value as other correct processes. We use this construction for Reliable Broadcast, but we believe it applies to other agreement problems in which all correct processes must decide.

The proof is still not immediate from here. In particular, since $f<n/2$, correct processes can wait until at least $f+1$ processes participate in each of E_v and E_w. Of those, in our proof we assume at most $f-1$ processes sign values. Since we need a bystander later, only $2f$ processes can participate. Thus, the sets executing E_v and E_w overlap at two processes; one must be the sender, to force decisions in both executions. Let p be the other process and S_v and S_w be the set that execute E_v and E_w respectively, without the sender and p. Thus, $|S_v| = |S_w| = f-1$.

The key complication is that if p signs its values in one of these two executions, we cannot compose them into an execution E in which the bystander b cannot distinguish which value it should decide. To see this, assume without loss of generality that p signs a value in execution E_w. To create E, we need the sender s and the set S_w to be Byzantine. The sender will produce signed versions of both v and w for the two sets to use, and S_w will pretend to execute E_w even though they observed that E_v was executed first. Since $|S_w| + |\{s\}| = f$, all other processes must be correct. In particular, p will be correct, and will not produce

the signature that it produces in E_w. Thus, the bystander b will know that S_v were correct. More generally, the problem is that, while we know that at most $f-1$ processes sign, we do not know *which* processes sign. A clever algorithm can choose signing processes to defeat the indistinguishability argument – in our case, this happens if p is a process that signs.

Due to this issue, we take a slightly different approach for the Reliable Broadcast lower bound, first using the bystander construction to show that any Reliable Broadcast algorithm must produce *a single non-sender* signature. To strengthen this to our bound, we construct an execution in which this signature needs to be repeatedly produced. To make this approach work, we show not just that *there exists* an execution in which a non-sender signature is produced, but that *for all* executions of a certain form, a non-sender signature is produced. This change in quantifiers requires care in the indistinguishability proof, and allows us to repeatedly apply the result to construct a single execution that produces many signatures.

4.2 Proofs

In all proofs in this section, we denote by s the designated sender process in the broadcast protocols we consider. We first show that Consistent Broadcast requires at least one signature.

▶ **Lemma 4.1.** *Any algorithm for Consistent Broadcast in the M&M model with $n = 2f+1$ and $f \geq 1$ has an execution in which at least one correct process signs.*

Proof. By contradiction, assume there is some algorithm A for Consistent Broadcast in the M&M model with $n = 2f+1$ and $f \geq 1$ without any correct process signing. Partition Π into 3 subsets: S_1, S_2, and $\{p\}$, where S_1 contains the sender, $|S_1| = f$, $|S_2| = f$, and p is a single process. Let v, w be two distinct messages. Consider the following executions.

Execution $E_{\text{CLEAN-V}}$. Processes in S_1 and p are correct (including the sender s), while processes in S_2 are faulty and never take a step. Initially, s broadcasts v. Since s is correct, processes in S_1 and p eventually deliver v. By our assumption that correct processes never sign, processes in S_1 and p do not sign in this execution; processes in S_2 do not sign either, because they do not take any steps.

Execution $E_{\text{DIRTY-W}}$. Processes in S_1 and S_2 are correct but p is Byzantine. Initially, p sends all messages and writes to shared memory as it did in $E_{\text{CLEAN-V}}$ (it does so without following its algorithm; p is able to do this since no process signed in $E_{\text{CLEAN-V}}$). Then, the correct sender s broadcasts w and processes in S_1 and S_2 execute normally, while p stops executing. Then, by correctness of the algorithm, eventually all correct processes deliver w. By our assumption that correct processes never sign, processes in S_1 and S_2 do not sign in this execution; p does not sign either, because it acts as it did in $E_{\text{CLEAN-V}}$.

Execution E_{BAD}. Processes in S_1 are Byzantine, while processes in S_2 and p are correct. Initially, processes in S_2 sleep, while processes in S_1 and p execute, where processes in S_1 send the same messages to p and write the same values to shared memory as in $E_{\text{CLEAN-V}}$ (but they do not send any messages to S_2), so that from p's perspective the execution is indistinguishable from $E_{\text{CLEAN-V}}$. S_1 are able to do this because no process signed in $E_{\text{CLEAN-V}}$. Therefore, p eventually delivers v. Next, processes in S_1 write the initial values to their registers[1]. Now, process p stops executing, while processes in S_1 and S_2 execute the same steps as in $E_{\text{DIRTY-W}}$ – here, note that S_2 just follows algorithm A while S_1 is Byzantine and pretends to be in an execution where s broadcasts w (S_1 is able to do this because no process

[1] Recall that registers are single-writer. By "their registers", we mean the registers to which the processes can write.

signed in $E_{\text{DIRTY-W}}$). Because this execution is indistinguishable from $E_{\text{DIRTY-W}}$ to processes in S_2, they eventually deliver w. At this point, correct process p has delivered v while processes in S_2 (which are correct) have delivered w, which contradicts the consistency property of Consistent Broadcast. ◀

An algorithm for Reliable Broadcast works for Consistent Broadcast, so Lemma 4.1 also applies to Reliable Broadcast.

We now show a separation between Consistent Broadcast and Reliable Broadcast: any algorithm for Reliable Broadcast has an execution where at least $f-1$ correct processes sign.

The proof for the Reliable Broadcast lower bound has two parts. First, we show that intuitively there are many executions in which some process produces a signature: if E is an execution in which (1) two processes never take steps, (2) the sender is correct, and (3) processes fail only by crashing, then some non-sender process signs. This is the heart of the proof, and relies on the indistinguishability arguments discussed in Section 4.1. Here, we focus only on algorithms in which at most f correct processes sign, otherwise the algorithm trivially satisfies our final theorem.

▶ **Lemma 4.2.** *Let A be an algorithm for Reliable Broadcast in the M&M model with $n = 2f + 1$ and $f \geq 2$ processes, such that in any execution at most f correct processes sign. In all executions of A in which at least 2 processes crash initially, processes fail only by crashing, and the sender is correct, at least one correct non-sender process signs.*

Proof. By contradiction, assume some algorithm A satisfies the conditions of the lemma, but there is some execution of A where the sender s is correct, processes fail only by crashing, and at least 2 processes crash initially, but no correct non-sender process signs. Let $E_{\text{CLEAN-V}}$ be such an execution, D be a set with two processes that crash initially in $E_{\text{CLEAN-V}}$[2], $C = \Pi \setminus D$, and v be the message broadcast by s in $E_{\text{CLEAN-V}}$. Consider the following executions:

Execution $E_{\text{CLEAN-W}}$. The sender s broadcasts some message $w \neq v$, D crashes initially, and C is correct. Since s is correct, eventually all correct processes deliver w. By assumption, at most f processes sign. Let $S \subset C$ contain all processes that sign, augmented with any other processes so that $|S| = f$. Let $T = C \setminus S$. Note that (1) $|T| = f - 1$ and (2) if s signed, then $s \in S$, otherwise $s \in T$.

Execution $E_{\text{CLEAN-V}}$. This execution was defined above (where s broadcasts v). Since s is correct, eventually all correct processes deliver v. At least one process in T is correct – call it p_t – since processes in D are faulty and there are at least $f + 1$ correct processes. Note that p_t delivers v. We refer to p_t in the next execution.

Execution $E_{\text{MIXED-V}}$. Processes in S are Byzantine and the rest are correct. Initially, the execution is identical to $E_{\text{CLEAN-V}}$, except that (1) processes in D are just sleeping not crashed, and (2) processes in S do not send messages to processes in D (this is possible because processes in S are Byzantine). The execution continues as in $E_{\text{CLEAN-V}}$ until p_t delivers v. Then, processes in S misbehave (they are Byzantine) and do three things: (1) they change their states to what they were at the end of $E_{\text{CLEAN-W}}$ (this is possible because no process in T signed in $E_{\text{CLEAN-W}}$), (2) they write to their registers in shared memory the same last values that they wrote in $E_{\text{CLEAN-W}}$, and (3) they send the same messages they did in $E_{\text{CLEAN-W}}$. Intuitively, processes in S pretend that s broadcast w. Let t be the time at this point; we refer to time t in the next execution. Now, we pause processes in S and let all other processes execute, including D which had been sleeping. Since p_t delivered v and processes in D are correct, they eventually deliver v as well.

[2] If more than two processes crashed initially, pick any two arbitrarily.

Execution E_{BAD}. Processes in $T \cup \{s\}$ are Byzantine and the rest are correct. Initially, the execution is identical to $E_{\text{CLEAN-W}}$, except that (1) processes in D are sleeping not crashed, and (2) processes in $T \cup \{s\}$ do not send messages to processes in D. Execution continues as in $E_{\text{CLEAN-W}}$ until processes in S (which are correct) deliver w. Then, processes in $T \cup \{s\}$ misbehave and do three things: (1) they change their states to what they were in $E_{\text{MIXED-V}}$ at time t – this is possible because in $E_{\text{CLEAN-V}}$ (and therefore in all values and messages they had by time t in $E_{\text{MIXED-V}}$), no non-sender process signed, and in particular, there were no signatures by any process in $S \setminus \{s\}$; (2) they write to the registers in shared memory the same values that they have in $E_{\text{MIXED-V}}$ at time t; and (3) they send all messages they did in $E_{\text{MIXED-V}}$ up to time t. Intuitively, processes in $T \cup \{s\}$ pretend that s broadcast v. Now, processes in D start executing. In fact, execution continues as in $E_{\text{MIXED-V}}$ from time t onward, where processes is S are paused and all other processes execute (including D). Because these processes cannot distinguish the execution from $E_{\text{MIXED-V}}$, eventually they deliver v. Note that processes in D are correct and they deliver v, while processes in S are also correct and deliver w – contradiction. ◀

In the final stage of the proof, we leverage Lemma 4.2 to construct an execution in which many processes sign. This is done by allowing some process to be poised to sign, and then pausing it and letting a new process start executing. Thus, we apply Lemma 4.2 $f - 1$ times to incrementally build an execution in which $f - 1$ correct processes sign.

▶ **Theorem 4.3.** *Any algorithm that solves Reliable Broadcast in the M&M model with $n = 2f + 1$, $f \geq 1$ has an execution in which at least $f-1$ correct non-sender processes sign.*

Proof. If $f = 1$, the result is trivial; it requires $f - 1 = 0$ processes to sign.

Now consider the case $f \geq 2$. If A has an execution in which at least $f + 1$ correct processes sign, then we are done. Now suppose A has no execution in which at least $f + 1$ correct processes sign. Consider the following execution of A.

All processes and s are correct. Initially, s broadcasts v. Then processes $s, p_1 \dots p_f$ participate, and the rest are delayed. This execution is indistinguishable to $s, p_1 \dots p_f$ from one in which the rest of the processes crashed. Therefore, by Lemma 4.2, some process in $p_1 \dots p_f$ eventually signs. Call p_1 the first process that signs. We continue the execution until p_1's next step is to make its signature visible. Then, we pause p_1, and let p_{f+1} begin executing. Again, this execution is indistinguishable to $s, p_2 \dots p_{f+1}$ from one in which the rest of the processes crashed, so by Lemma 4.2, eventually some process in $p_2 \dots p_{f+1}$ creates a signature and makes it visible. We let the first process to do so reach the state in which it is about to make its signature visible, and then pause it, and let p_{f+2} start executing.

We continue in this way, each time pausing p_i as it is about to make its signature visible, and letting p_{f+i} begin executing. We can apply Lemma 4.2 as long as two processes have not participated yet. At that point, $f - 1$ processes are poised to make their signatures visible. We then let these $f - 1$ processes each take one step. This yields an execution of A in which $f - 1$ correct non-sender processes sign. ◀

5 Broadcast Algorithms

In this section we present solutions for Consistent and Reliable Broadcast. We first implement Consistent Broadcast in Section 5.1; then we use it as a building block to implement Reliable Broadcast, in Section 5.2. We prove the correctness of our algorithms in the full version of our paper [4]. For both algorithms, we first describe the general execution outside the common case, which captures behavior in the worst executions; we then describe how delivery happens fast in the common case (without signatures).

Process roles in broadcast. We distinguish between three process roles in our algorithms: sender, receiver, and replicator. This is similar in spirit to the proposer-acceptor-learner model used by Paxos [38]. Any process may play any number of roles; if all processes play all three roles, then this becomes the standard model. The sender calls *broadcast*, the receivers call *deliver*, and the replicators help guarantee the properties of broadcast. By separating replicators (often servers) from senders and receivers (often clients or other servers), we improve the practicality of the algorithms: clients, by not fulfilling the replicator role, need not remain connected to disseminate information from other clients. Unless otherwise specified, n and f refer only to replicators; independently, the sender and any number of receivers can also be Byzantine. Receivers cannot send or write any values, as opposed to the sender and replicators, but they can read the shared memory and receive messages.

Background signatures. Our broadcast algorithms produce signatures in the background. We do so to allow the algorithms to be signature-free in the common case. Indeed, in the common case, receivers can deliver a message without waiting for background signatures. However, outside the common case, these signatures must still be produced by the broadcast algorithms in case some replicators are faulty or delayed. Both algorithms require a number of signatures that matches the bounds in Section 4 within constant factors.

5.1 Consistent Broadcast

We give an algorithm for Consistent Broadcast that issues no signatures in the common case, when there is synchrony and no replicator is faulty. Outside this case, only the sender signs.

Algorithm 1 shows the pseudocode. The broadcast and deliver events are called *cb-broadcast* and *cb-deliver*, to distinguish them from *rb-broadcast* and *rb-deliver* of Reliable Broadcast. Processes communicate by sharing an array of *slots*: process i can write to *slots[i]*, and can read from all slots. To refer to its own slot, a processes uses index *me*. The sender s uses its slot to broadcast its message while replicators use their slot to replicate the message. Every slot has two sub-slots – each a SWMR atomic register – one for a message (*msg*) and one for a signature (*sgn*).

To broadcast a message m, the sender s writes m to its *msg* sub-slot (line 6). Then, in the background, s computes its signature for m and writes it to its *sgn* sub-slot (line 9). The presence of *msg* and *sgn* sub-slots allow the sender to perform the signature computation in the background. Sender s can return from the broadcast while this background task executes.

The role of a correct replicator is to copy the sender's message m and signature σ, provided σ is valid. The copying of m and σ (lines 12–19) are independent events, since a signature may be appended in the background, i.e., later than the message. The fast way to perform a delivery does not require the presence of signatures. Note that correct replicators can have mismatching values only when s is Byzantine and overwrites its memory.

A receiver p scans the slots of the replicators. It delivers message m when the content of a majority $(n-f)$ of replicator slots contains m and a valid signature by s for m, and no slot contains a different message $m', m' \neq m$ with a valid sender signature (line 28). Slots with sender signatures for $m' \neq m$ result in a no-delivery. This scenario indicates that the sender is Byzantine and is trying to equivocate. Slots with signatures not created by s are ignored so that a Byzantine replicator does not obstruct p from delivering.

When there is synchrony and both the sender and replicators follow the protocol, a receiver delivers without using signatures. Specifically, delivery in the fast path occurs when there is unanimity, i.e., all $n = 2f + 1$ replicators replicated value m (line 25), regardless

Algorithm 1 Consistent Broadcast Algorithm with sender s.

```
1  Shared:
2  slots - n array of "slots"; each slot is a 2-tuple (msg, sgn) of SWMR atomic registers, initialized
       ↪ to (⊥, ⊥).

4  Sender code:
5  cb-broadcast(m):
6      slots[me].msg.write(m)
7      In the background:
8          σ = compute signature for m
9          slots[me].sgn.write(σ)

11 Replicator code:
12 while True:
13     m = slots[s].msg.read()
14     if m ≠ ⊥:
15         slots[me].msg.write(m)
16     sign = slots[s].sgn.read()
17     val = slots[me].msg.read()
18     if val ≠ ⊥ and sign ≠ ⊥ and sign is a valid signature for val:
19         slots[me].sgn.write(sign)

21 Receiver code:
22 while True:
23     others = scan()
24     if others[i].msg has the same value m for all i in Π: // Fast path
25         cb-deliver(m); break
26     if others contains at least n − f signed copies of the same value m
27         and (∄i: others[i].sgn is a valid signature for others[i].msg and others[i].msg ≠ m):
28         cb-deliver(m); break

30 scan():
31     others = [slots[i].(msg, sgn).read() for i in Π]
32     done = False
33     while not done:
34         done = True
35         for i in Π:
36             if others[i] == ⊥:
37                 others[i] = slots[i].(msg, sgn).read()
38                 if others[i] ≠ ⊥:
39                     done = False
40     return others
```

of whether a signature is provided by s. A correct sender eventually appends σ, and $n - f$ correct replicators eventually copy σ over, allowing another receiver to deliver m via the slow path, even if a replicator misbehaves, e.g., removes or changes its value.

An important detail is the use of a snapshot to read replicators' slots (line 23), as opposed to a simple collect. The scan operation is necessary to ensure that concurrent reads of the replicators' slots do not return views that can cause correct receivers to deliver different messages. To see why, imagine that the scan at line 23 is replaced by a simple collect. Then, an execution is possible in which correct receiver p_1 reads some (correctly signed) message m_1 from $n - f$ slots and finds the remaining slots empty, while another correct receiver p_2 reads $m_2 \neq m_1$ from $n - f$ slots and finds the remaining slots empty. In this execution, p_1 would go on to deliver m_1 and p_2 would go on to deliver m_2, thus breaking the consistency property. We present such an execution in detail in [4].

To prevent scenarios where correct receivers see different values at a majority of replicator slots, the *scan* operation works as follows (lines 30–40): first, it performs a collect of the slots. If all the slots are non-empty, then we are done. Otherwise, we re-collect the *empty slots* until no slot becomes non-empty between two consecutive collects. This suffices to avoid the problematic scenario above and to guarantee liveness despite f Byzantine processes.

5.2 Reliable Broadcast

We now give an algorithm for Reliable Broadcast that issues no signatures in the common case, and issues only $n + 1$ signatures in the worst case. Algorithm 3 (part of Appendix B) shows the pseudocode.

Processes communicate by sharing arrays *Echo* and *Ready*, which have the same structure of sub-slots as *slots* in Section 5.1. *Echo[i]* and *Ready[i]* are writable only by replicator i, while the sender s communicates with the replicators using an instance of Consistent Broadcast (CB) and does not access *Echo* or *Ready*. In this CB instance, s invokes *cb-broadcast*, acting as sender for CB, and the replicators invoke *cb-deliver*, acting as receivers for CB.

To broadcast a message, s *cb-broadcast*s $\langle$INIT,$m\rangle$ (line 6). Upon delivering the sender's message $\langle$INIT,$m\rangle$, each replicator writes m to its *Echo msg* sub-slot (line 13). Then, in the background, a replicator computes its signature for m and writes it to its *Echo sgn* sub-slot (line 16). By the consistency property of Consistent Broadcast, if two correct replicators r and r' deliver $\langle$INIT,$m\rangle$ and $\langle$INIT, $m'\rangle$ respectively, from s, then $m = m'$. Essentially, correct replicators have the same value or $\perp$ in their *Echo msg* sub-slot.

Next, replicators populate their *Ready* slots with a *ReadySet*. A replicator r constructs such a *ReadySet* from the $n - f$ signed copies of m read from the *Echo* slots (lines 19–28). In the background, r reads the *Ready* slots of other replicators and copies over – if r has not written one already – any valid *ReadySet* (line 36). Thus, totality is ensured (Definition 3.2), as the *ReadySet* created by any correct replicator is visible to all correct receivers.

To deliver m, a receiver p reads $n - f$ valid *ReadySet*s for m (line 45).[3] This is necessary to allow a future receiver p' deliver a message as well. Suppose that p delivers m by reading a single valid *ReadySet* R.[4] Then, the following scenario prevents p' from delivering: let sender s be Byzantine and let R be written by a Byzantine replicator r. Moreover, let a *single* correct replicator have *cb-deliver*ed m, while the remaining correct replicators do not deliver at all, which is allowed by the properties of Consistent Broadcast. So, the *ReadySet* contains values from a single correct replicator and f other Byzantine replicators. If r removes R from its *Ready* slot, it will block the delivery for p' since no valid *ReadySet* exists in memory.

A receiver p can also deliver the sender's message m using a fast path. The signature-less fast path occurs when p reads m from the *Echo* slots of all replicators (line 43), and the delivery of the INIT message by the replicators is done via the fast path of Consistent Broadcast. This is the common case, when replicators are not faulty and replicate messages timely. Note that p delivering m via the fast path does not prevent another receiver p' from delivering. Process p' delivers m via the fast path if all the *Echo* slots are in the same state as for p. Otherwise, e.g., some Byzantine replicators overwrite their *Echo* slots, p' delivers m by relying on the $n - f$ correct replicators following the protocol (line 45).

6 Consensus

We now give an algorithm for consensus using Consistent Broadcast as its communication primitive, rather than the commonly used primitive, Reliable Broadcast. Our algorithm is based on the PBFT algorithm [20, 21] and proceeds in a sequence of (consecutive) views. It has four features: (1) it works for $n = 2f + 1$ processes, (2) it issues no signatures in the common case, (3) it issues $O(n^2)$ signatures on a view-change and (4) it issues $O(n)$ required background signatures within a view.

[3] In contrast to Algorithm 1, receivers need not use the *scan* operation when gathering information from the replicators' *Ready* slots because there can only be a single value with a valid *ReadySet*.

[4] A similar argument that breaks totality applies if p were to deliver m by reading $n - f$ signed values of m in the replicators' *Echo* slots.

Our algorithm uses a sequence of Consistent Broadcast instances indexed by a broadcast sequence number k. When process p broadcasts its k^{th} message m, we say that p broadcasts (k, m). We assume the following ordering across instances, which can be trivially guaranteed: (**FIFO delivery**) For $k \geq 1$, no correct process delivers (k, m_k) from p unless it has delivered (i, m_i) from p, for all $i < k$.

Algorithm 2 shows the pseudocode. The full version of our paper [4] has its correctness proof. The protocol proceeds in a sequence of consecutive *views*. Each view has a primary process, defined as the view number mod n (line 6). A view has two phases, PREPARE and COMMIT. There is also a view-change procedure initiated by a VIEWCHANGE message.

When a process is the primary (line 9), it broadcasts a PREPARE message with its estimate *init* (line 11), which is either its input value or a value acquired in the previous view (line 10). Upon receiving a valid PREPARE message, a replica broadcasts a COMMIT message (line 20) with the estimate it received in the PREPARE message. We define a PREPARE to be valid when it originates from the primary and either (a) $view = 0$ (any estimate works), or (b) $view > 0$ and the estimate in the PREPARE message has a proof from the previous view. The extended paper [4] details the conditions for a message to be valid. When a replica receives an invalid PREPARE message from the primary or times out, it broadcasts a COMMIT message with $\perp$. If a replica accepts a PREPARE message with *val* as estimate and $n - f$ matching COMMIT messages (line 24), it decides on *val*.

Algorithm 2 Consensus protocol based on Consistent Broadcast ($n = 2f + 1$)

```
1  propose(v_i):
2      view_i = 0; est_i = ⊥; aux_i = ⊥
3      proof_i = ∅; vc_i= (0, ⊥, ∅)
4      decided_i = False
5      while True:
6          p_i = view_i % n

8          // Phase 1
9          if p_i == i:
10             init_i = est_i if est_i ≠ ⊥ else v_i
11             cb-broadcast(⟨PREPARE, view_i, init_i, proof_i⟩)
12         wait until receive valid ⟨PREPARE, view_i, val, proof⟩ from p_i or timeout on p_i
13         if received valid ⟨PREPARE, view_i, val, proof⟩ from p_i:
14             aux_i = val
15             vc_i = (view_i,val,proof)
16         else:
17             aux_i = ⊥

19         // Phase 2
20         cb-broadcast(⟨COMMIT, view_i, aux_i⟩)
21         wait until receive valid ⟨COMMIT, view_i, *⟩ from n − f processes
22                  and (∀j: receive valid ⟨COMMIT, view_i, *⟩ from j or timeout on j)
23         ∀j: R_i[j] = val if received valid ⟨COMMIT, view_i, val⟩ from j else ⊥
24         if ∃val ≠ ⊥ : #_val(R_i) ≥ n − f and aux_i == val:
25             try_decide(val)

27         // Phase 3
28         cb-broadcast(⟨VIEWCHANGE, view_i + 1, vc_i⟩_σi)
29         wait until receive n − f non-conflicting view-change certificates for view_i + 1
30         proof_i = set of non-conflicting view-change certificates
31         est_i = val in proof_i associated with the highest view
32         view_i = view_i + 1

34     In the background:
35         when cb-deliver valid ⟨VIEWCHANGE, view', vc⟩_σj from j:
36             cb-broadcast(⟨VIEWCHANGEACK, d⟩_σi)  // d is the view-change message being ACKed

38 try_decide(val):
39     if not decided_i:
40         decided_i = True
41         decide(val)
```

The view-change procedure ensures that all correct replicas eventually reach a view with a correct primary and decide. It uses an acknowledgement phase similar to PBFT with MACs [21]. While in [21] the mechanism is used so that the primary can prove the authenticity of a view-change message sent by a faulty replica, we use this scheme to ensure that (a) a faulty participant cannot lie about a committed value in its ViewChange message and (b) valid ViewChange messages can be received by all correct replicas.

A replica starts a view-change by broadcasting a signed ViewChange message with its view-change tuple (line 28). The view-change tuple *(view, val, proof$_{val}$)* is updated when a replica receives a valid Prepare message (line 15). It represents the last non-empty value a replica accepted as a valid estimate and the view when this occurred. We use the value's proof, *proof$_{val}$*, to prevent a Byzantine replica from lying about its value: suppose a correct replica decides *val* in view v, but in view $v+1$, the primary p is silent, and so no correct replica hears from p; without the proof, a Byzantine replica could claim to have accepted *val′* in $v+1$ from p during the view-change to $v+1$, thus overriding the decided value *val*.

When a replica receives a valid ViewChange message, it responds by broadcasting a signed ViewChangeAck containing the ViewChange message (line 36). A common practice is to send a digest of this message instead of the entire message [20]. We define a ViewChange message m from p to be valid when the estimate in the view-change tuple corresponds to the value broadcast by p in its latest non-empty Commit and m's proof is valid. We point out that, as an optimization, this proof can be removed from the view-change tuple and be provided upon request when required to validate ViewChange messages. For instance, in the scenario described above, when a (correct) replica r did not accept *val′* in view $v+1$, as claimed by the Byzantine replica r', r can request r' to provide a proof for *val′*.

A view-change certificate consists of a ViewChange message and $n-f-1$ corresponding ViewChangeAck messages. This way, each view-change certificate has the contribution of at least one correct replica, who either produces the ViewChange message or validates a ViewChange message. Thus, when a correct replica r receives a view-change certificate relayed by the primary, r can trust the contents of the certificate.

To move to the next view, a replica must gather a set of $n-f$ non-conflicting view-change certificates Ψ. This step is performed by the primary of the next view, who then includes this set with its Prepare message for the new view. Two view-change certificates conflict if their view-change messages carry a tuple with different estimates ($\neq \perp$), valid proof, and same view number. If the set Ψ consists of tuples with estimates from different views, we select the estimate associated with the highest view. Whenever any correct replica decides on a value *val* within a view, the protocol ensures a set of non-conflicting view-change certificates can be constructed only for *val* and hence the value is carried over to the next view(s).

Discussion

We discuss how Algorithm 2 achieves the four features mentioned at the beginning of Section 6. The first feature (the algorithm solves consensus with $n = 2f + 1$ processes) follows directly from the correctness of the algorithm. The second feature (the algorithm issues no signatures in the common case) holds because in the common case, processes will be able to deliver the required Prepare and Commit messages and decide in the first view, without having to wait for any signatures to be produced or verified. The third feature (the algorithm issues $O(n^2)$ signatures on view-change) holds because, in the worst case, during a view change each process will sign and broadcast a ViewChange message, thus incurring $O(n)$ signatures in total, and, for each such message, each other process will sign and broadcast a ViewChangeAck message, thus incurring $O(n^2)$ signatures. The fourth feature states that

the algorithm issues $O(n)$ required background signatures within a view. These signatures are incurred by *cb-broadcast*ing PREPARE and COMMIT messages. In every view, correct processes broadcast a COMMIT message, thus incurring $n - f = O(n)$ signatures in total.

To the best of our knowledge, no existing algorithm has achieved all these four features simultaneously. The only broadcast-based algorithm which solves consensus with $n = 2f + 1$ processes that we are aware of, that of Correia et al. [27], requires $O(n)$ calls to Reliable Broadcast before any process can decide; this would incur $O(n^2)$ required background signatures when using our Reliable Broadcast implementation – significantly more than our algorithm's $O(n)$ required background signatures.

At this point, the attentive reader might have noticed that our consensus algorithm uses some techniques that bear resemblance to our Reliable Broadcast algorithm in Section 5. Namely, the primary of a view *cb-broadcast*s a PREPARE message which is then echoed by the replicas in the form of COMMIT messages. Also, during view change, a replica's VIEWCHANGE message is echoed by other replicas in the form of VIEWCHANGEACK messages. This is reminiscent of the Init-Echo technique used by our Reliable Broadcast algorithm.

Thus, the following question arises: Can we replace each instance of the witnessing technique in our algorithm by a single Reliable Broadcast call and thus obtain a conceptually simpler algorithm, which also satisfies the three above-mentioned properties? Perhaps surprisingly, the resulting algorithm is incorrect. It allows an execution which breaks agreement in the following way: a correct replica p_1 *rb-delivers* some value v from the primary and decides v; sufficiently many other replicas time out waiting for the primary's value and change views without "knowing about" v; in the next view, the primary *rb-broadcasts* v', which is delivered and decided by some correct replica p_2.

Intuitively, by using a single Reliable Broadcast call instead of multiple Consistent Broadcast calls, some information is not visible to the consensus protocol. Specifically: while it is true that, in order for p_1 to deliver v in the execution above, $n - f$ processes must echo v (and thus they "know about" v), this knowledge is however encapsulated inside the Reliable Broadcast abstraction and not visible to the consensus protocol. Thus, the information cannot be carried over to the view-change, even by correct processes. This intuition provides a strong motivation to use Consistent Broadcast – rather than Reliable Broadcast – as a first-class primitive in the design of Byzantine-resilient agreement algorithms.

7 Conclusion

A common tool to address Byzantine failures is to use signatures or lots of replicas. However, modern hardware makes these techniques prohibitive: signatures are much more costly than network communication, and excessive replicas are expensive. Hence, we seek algorithms that minimize the number of signatures and replicas. We applied this principle to broadcast primitives in the message-and-memory model, and derived algorithms that avoid signatures in the common case, use nearly-optimal number of signatures in the worst case, and require only $n = 2f+1$ replicas. We proved worst-case lower bounds on the number of signatures required by Consistent Broadcast and Reliable Broadcast, showing a separation between these problems. We presented the first Byzantine consensus algorithm for $n = 2f+1$ without signatures in the common case. A novelty of our protocol is the use of Consistent Broadcast instead of Reliable Broadcast, which resulted in fewer signatures than existing consensus protocols based on Reliable Broadcast.

References

1. Marcos K Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, 2018.
2. Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of RDMA on agreement. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 409–418, 2019.
3. Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 599–616, 2020.
4. Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Dalia Papuc, Athanasios Xygkis, and Igor Zablotchi. Frugal Byzantine Computing. *arXiv preprint*, 2021. `arXiv:2108.01330`.
5. Noga Alon, Michael Merritt, Omer Reingold, Gadi Taubenfeld, and Rebecca N Wright. Tight bounds for shared memory systems accessed by Byzantine processes. *Distributed computing (DIST)*, 18(2), 2005.
6. Hagit Attiya, Sweta Kumari, and Noa Schiller. Optimal resilience in systems that mix shared memory and message passing. *arXiv preprint*, 2020. `arXiv:2012.10846`.
7. Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4), 2015.
8. Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 349–359, 2016.
9. Naama Ben-David and Kartik Nayak. Brief announcement: Classifying trusted hardware via unidirectional communication. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2021.
10. Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, 1983.
11. Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. Sharing memory between Byzantine processes using policy-enforced tuple spaces. *IEEE Transactions on Parallel and Distributed Systems*, 20(3), 2009.
12. Bitcoin Core Developers. Optimized C library for ECDSA signatures and secret/public key operations on curve secp256k1. `https://github.com/bitcoin-core/secp256k1`.
13. Zohir Bouzid, Damien Imbs, and Michel Raynal. A necessary condition for Byzantine k-set agreement. *Information Processing Letters*, 116(12), 2016.
14. Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 154–162, 1984.
15. Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2), 1987.
16. Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 12–26, 1983.
17. Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4), 1985.
18. Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.
19. Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 2001.
20. Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 1999.

21 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002. doi:10.1145/571637.571640.

22 Certicom Research. Standards for efficient cryptography. https://www.secg.org/sec2-v2.pdf, 2010.

23 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2), 1996.

24 Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse replication for single-machine Byzantine-fault tolerance. In *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings*, 2008. URL: http://www.usenix.org/events/usenix08/tech/full_papers/chun/chun.pdf.

25 Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, 2007. doi:10.1145/1294261.1294280.

26 Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil*, 2004. doi:10.1109/RELDIS.2004.1353018.

27 Miguel Correia, Giuliana S Veronese, and Lau Cheuk Lung. Asynchronous Byzantine consensus with $2f + 1$ processes. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.

28 Dan Dobre and Neeraj Suri. One-step consensus with zero-degradation. In *International Conference on Dependable Systems and Networks (DSN'06)*, 2006.

29 Danny Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

30 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.

31 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2), 1988.

32 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 1985.

33 Vassos Hadzilacos, Xing Hu, and Sam Toueg. Optimal Register Construction in M&M Systems. In *International Conference on Principles of Distributed Systems (OPODIS)*, volume 153, pages 28:1–28:16, 2020.

34 Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: resource-efficient Byzantine fault tolerance. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, 2012. doi:10.1145/2168836.2168866.

35 Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults: preliminary version. *ACM SIGACT News*, 32(2), 2001.

36 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

37 Leslie Lamport. The weak Byzantine generals problem. *Journal of the ACM (JACM)*, 30(3), 1983.

38 Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), 1998.

39 Leslie Lamport. Fast Paxos. *Distributed computing (DIST)*, 19(2), 2006.

40 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3), 1982.

41 linux-rdma. Rdma benchmarking utility. https://github.com/linux-rdma/perftest.

42 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

43 Dahlia Malkhi, Michael Merritt, Michael K Reiter, and Gadi Taubenfeld. Objects shared by Byzantine processes. *Distributed computing (DIST)*, 16(1), 2003.

44 Mellanox. Network benchmarking utility. `https://github.com/Mellanox/sockperf`.

45 Achour Mostéfaoui, Moumen Hamouma, and Michel Raynal. Signature-free asynchronous Byzantine consensus with $t < n/3$ and $o(n^2)$ messages. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 2–9, 2014.

46 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2), 1980.

47 Michel Raynal and Jiannong Cao. One for all and all for one: Scalable consensus in a hybrid communication model. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 464–471. IEEE, 2019.

48 Michael K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68—-80, 1994. `doi:10.1145/191177.191194`.

49 Blockchain hardware accelerator. `https://www.xilinx.com/products/intellectual-property/1-175rk99.html`. Accessed 2021-02-15.

50 T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed computing (DIST)*, 2(2):80–94, 1987.

51 Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 49(8):357–368, 2014.

52 Sam Toueg. Randomized Byzantine agreements. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 163–178, 1984.

53 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient Byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1), 2013. `doi:10.1109/TC.2011.221`.

A APPENDIX: Latency

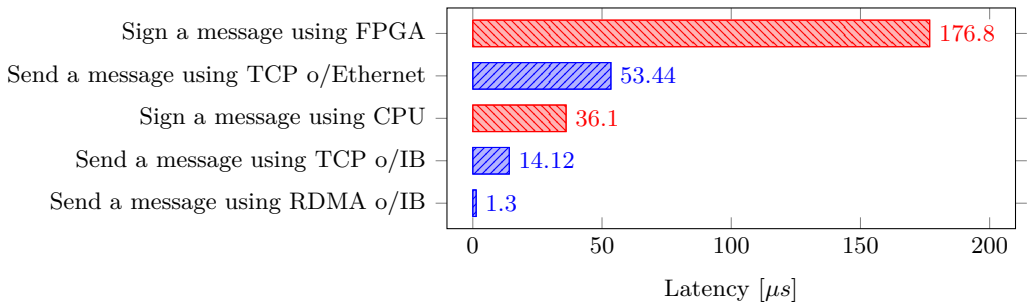

Figure 1 RDMA communication is significantly faster than signature creation using CPU or hardware acceleration (FPGA). The graph shows the latency of sending or signing a 32-byte message. IB means Infiniband, a faster interconnect than Ethernet found in data centers. TCP latencies are obtained using sockperf [44]. RDMA latency is obtained using perftest [41]. Signatures use optimized implementations for CPU [12] and FPGA [49] of the ECDSA algorithm on the secp256k1 elliptic curve [22]. An FPGA improves the throughput of signature creation (not shown in figure), but not its latency, due to their relatively low clock speeds (compared to CPUs) and the non-parallelizable nature of algorithms for digital signature.

B APPENDIX: Reliable Broadcast Algorithm

Algorithm 3 Reliable Broadcast Algorithm with sender s.

```
1  Shared:
2  Echo, Ready - n array of "slots"; each slot is a 2-tuple (msg, sgn) of SWMR atomic registers,
       ↪ initialized to (⊥, ⊥).

4  Sender code:
5  rb-broadcast(m):
6      cb-broadcast(⟨INIT,m⟩)

8  Replicator code:
9  state = WaitForSender // ∈{WaitForSender,WaitForEchos}
10 while True:
11     if state == WaitForSender:
12         if cb-delivered ⟨INIT,m⟩ from s:
13             Echo[me].msg.write(m)
14             In the background:
15                 σ = compute signature for m
16                 Echo[me].sgn.write(σ)
17             state = WaitForEchos

19     if state == WaitForEchos:
20         ReadySet = ∅
21         for i ∈ Π:
22             other = Echo[i].(msg,sgn).read()
23             if other.msg == m and other.sgn is m validly signed by i:
24                 ReadySet.add((i,other))

26         if size(ReadySet) ≥ n − f:
27             ready = True
28             Ready[me].msg.write(ReadySet)

30 In the background:
31     while True
32         if not ready:
33             others = [Ready[i].msg.read() for i in Π]
34             if ∃i: others[i] is a valid ReadySet:
35                 ready = True
36                 Ready[me].msg.write(others[i])

38 Receiver code:
39 while True:
40     others = [Echo[i].msg.read() for i in Π]
41     proofs = [Ready[i].msg.read() for i in Π]
42     if others contains n matching values m: // Fast path
43         rb-deliver(m); break
44     if proofs contains n − f valid ReadySet for the same value m:
45         rb-deliver(m); break
```