Parsimonious Asynchronous Byzantine-Fault-Tolerant Atomic Broadcast

HariGovind V. Ramasamy[*] Christian Cachin[†]

August 19, 2005

Abstract

Atomic broadcast is a communication primitive that allows a group of n parties to deliver a common sequence of payload messages despite the failure of some parties. We address the problem of asynchronous atomic broadcast when up to $t < n/3$ parties may exhibit Byzantine behavior. We provide the first protocol with an amortized expected message complexity of $\mathcal{O}(n)$ per delivered payload. The most efficient previous solutions are the BFT protocol by Castro and Liskov and the KS protocol by Kursawe and Shoup, both of which have message complexity $\mathcal{O}(n^2)$. Like the BFT and KS protocols, our protocol is optimistic and uses inexpensive mechanisms during periods when no faults occur; when network instability or faults are detected, it switches to a more expensive recovery mode. The key idea of our solution is to replace reliable broadcast in the KS protocol by consistent broadcast, which reduces the message complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ in the optimistic mode. But since consistent broadcast provides weaker guarantees than reliable broadcast, our recovery mode incorporates novel techniques to ensure that safety and liveness are always satisfied.

1 Introduction

Atomic broadcast is a fundamental communication primitive for the construction of fault-tolerant distributed systems. It allows a group of n parties to agree on a set of payload messages to deliver and also on their delivery order, despite the failure of up to t parties. A fault-tolerant service can be constructed using the state machine replication approach [22] by replicating the service on all n parties and propagating the state updates to the replicas using atomic broadcast.

In this paper, we present a new message-efficient atomic broadcast protocol that is suitable for building highly available and intrusion-tolerant services in the Internet [4][23]. Since the Internet is an adversarial environment where an attacker can compromise and completely take over nodes, we allow the corrupted parties to deviate arbitrarily from the protocol specification thereby exhibiting so-called *Byzantine faults*. We work in an asynchronous system model for two reasons: (1) it best reflects the loosely synchronized nature of nodes in the Internet, and (2) not relying on synchrony assumptions for correctness also eliminates a potential vulnerability of the system that the adversary can exploit, for example, through denial-of-service attacks.

Though the problem of Byzantine-fault-tolerant atomic broadcast and the equivalent problem of Byzantine agreement have been widely studied for over two decades, the applicability of many of the previous works for our purpose is quite limited. Any asynchronous atomic broadcast protocol must use randomization, since deterministic solutions cannot be guaranteed to terminate [10]. Early work focused on the polynomial-time feasibility of randomized agreement [18][7][2] and atomic broadcast [1], but such solutions are too expensive to use in practice. Many protocols have followed an alternative approach and avoided randomization completely by making stronger assumptions about the system

[*]H. V. Ramasamy is with the University of Illinois, Urbana-Champaign. Email: `ramasamy@crhc.uiuc.edu`.
[†]C. Cachin is with the IBM Zurich Research Laboratory. Email: `cca@zurich.ibm.com`.

model, in particular by assuming some degree of synchrony (like Rampart [21], SecureRing [14], and ITUA [19]). However, most of these protocols have an undesirable feature that makes them inapplicable for our purpose: they may violate safety if synchrony assumptions are not met.

Only recently, Cachin et al. proposed practical asynchronous agreement [6] and atomic broadcast [5] protocols that have optimal resilience $t < n/3$. Both protocols rely on a trusted initialization process and on public-key cryptography. Cachin et al.'s atomic broadcast protocol proceeds in rounds, with each round involving a randomized Byzantine agreement and resulting in the atomic delivery of some payload messages.

The BFT protocol by Castro and Liskov [8] and the protocol by Kursawe and Shoup [15] (hereafter referred to as the KS protocol) take an optimistic approach for providing more efficient asynchronous atomic broadcast while never violating safety. The motivation for such optimistic protocols is the observation that conditions are *normal* during most of a system's operation. Here, normal conditions refer to a stable network and no intrusions. Both protocols proceed in *epochs*, where an epoch consists of an *optimistic phase* and a *recovery phase*, and expect to spend most of their time operating in the optimistic phase, which uses an inexpensive mechanism that is appropriate for normal conditions. The protocol switches to the more expensive recovery phase under unstable network or certain fault conditions. In every epoch, a designated party acts as a *leader* for the optimistic phase, determines the delivery order of the payloads, and conveys the chosen delivery order to the other parties through Bracha's reliable broadcast protocol [3], which guarantees delivery of a broadcast payload with the same content at all correct parties. Bracha's protocol is deterministic and involves $\mathcal{O}(n^2)$ protocol messages; it is much more efficient than the most efficient randomized Byzantine agreement protocol [5], which requires expensive public-key cryptographic operations in addition. Consequently, both the BFT and KS protocols communicate $\mathcal{O}(n^2)$ messages per atomically delivered payload under normal conditions, i.e., they have *message complexity* $\mathcal{O}(n^2)$.

No protocol for asynchronous atomic broadcast with message complexity less than $\Theta(n^2)$ was known prior to our work. Our protocol for asynchronous atomic broadcast is the first to achieve optimal resilience $t < n/3$ and $\mathcal{O}(n)$ amortized expected message complexity. We call our protocol *parsimonious* because of this significant reduction in message complexity. Linear message complexity appears to be optimal for atomic broadcast because a protocol needs to send every payload to each party at least once and this requires n messages (assuming that payloads are not propagated to the parties in batches). Like the BFT and KS protocols, our protocol is *optimistic* in the sense that it progresses very fast during periods when the network is reasonably behaved and a party acting as designated *leader* is correct. Unlike the BFT protocol (and just like the KS protocol), our protocol guarantees both *safety* and *liveness* in asynchronous networks by relying on randomized agreement. The reduced message complexity of our protocol comes at the cost of introducing a digital signature computation for every delivered payload. But in a wide-area network (WAN), the cost of a public-key operation is small compared to message latency. And since our protocol is targeted at WANs, we expect the advantage of lower message complexity to outweigh the additional work incurred by the signature computations.

The key idea in our solution is to replace reliable broadcast used in the optimistic phase of the BFT and KS protocols with *consistent broadcast*, also known as *echo broadcast* [20], the standard implementation of which needs only $\mathcal{O}(n)$ messages. Consistent broadcast is a weaker form of reliable broadcast that guarantees agreement only among those correct parties that actually deliver the payload, but it is possible that some correct parties do not deliver any payload at all. But the replacement also complicates the recovery phase, since a corrupted leader might cause the payload to be consistently delivered at only a single correct party with no way for other correct parties to learn about this fact. Our protocol provides mechanisms to address such complications.

Our protocol is related to the reliable broadcast protocol of Malkhi et al. [17] in its use of consistent broadcast as a building block. Their protocol addresses reliable broadcast over a WAN, but provides no total order.

The rest of the paper is organized as follows. Section 2 describes the formal system model, the protocol primitives on which our algorithm relies, and the definition of atomic broadcast. The protocol is presented in Section 3 and analyzed in Section 4. Section 5 discusses the practical significance of our parsimonious protocol and compares it with related work. Finally, Section 6 concludes the paper.

2 Preliminaries

2.1 System Model

We consider an asynchronous distributed system model equivalent to the one of Cachin et al. [5], in which there are no bounds on relative processing speeds and message delays. The system consists of n *parties* $P_1, \ldots, P_n$ and an *adversary*. Up to $t < n/3$ parties can be controlled by the adversary. We call such parties *corrupted*; the other parties are called *correct*. We use a *static* corruption model, which means that the adversary must pick the parties it corrupts once and for all before starting the protocol. There is also an initialization algorithm that is run by some trusted *dealer* that performs system setup before the start of the protocol. All computations by the parties, the adversary, and the trusted dealer are probabilistic, polynomial-time algorithms. The parameters n and t are given as input to the dealer, which then generates the state information that is used to initialize each party. Note that after the initial setup phase, the protocol has no need for the dealer.

Each pair of parties is linked by an *authenticated asynchronous channel* that provides message integrity (e.g., using message authentication codes [25]). The adversary determines the scheduling of messages on all the channels. Timeouts are messages that a party sends to itself; hence, the adversary controls the timeouts as well.

We restrict the adversary such that every run of the system is *complete*, i.e., every message sent by a correct party and addressed to a correct party is delivered unmodified before the adversary terminates[1]. We refer to this property in liveness conditions when we say that a message is *eventually* delivered or that a protocol instance *eventually* terminates.

There may be multiple protocol instances that are concurrently executing at each party. A protocol instance is invoked either by a higher-level protocol instance or by the adversary. Every protocol instance is identified by a unique string ID, called the *tag*, which is chosen by the entity that invokes the instance. By convention, the tag of a sub-protocol instance contains the tag of the calling instance as a prefix.

A correct party is activated when the adversary delivers a message to the party; the party then updates its internal state, performs some computation, and generates a set of response messages, which are given to the adversary. There may be several threads of execution for a given party, but only one of them is allowed to be active at any one time. When a party is activated, all threads are in *wait states*, which specify a condition defined on the received messages contained in the input buffer, as well as on some local variables. In the pseudocode presentation of the protocol, we specify a wait state using the notation **wait for** *condition*. There is a global implicit **wait for** statement that every protocol instance repeatedly executes: it matches any of the *conditions* given in the clauses of the form **upon** *condition block*. If one or more threads that are in wait states have their conditions simultaneously satisfied, one of these threads is scheduled (arbitrarily), and this thread runs until it reaches another wait state. This process continues until no more threads are in a wait state whose condition is satisfied. Then, the activation of the party is terminated, and control returns to the adversary.

There are three types of messages that appear in the interface to our protocols, namely (1) *input actions*, which are messages of the form $(ID, \texttt{in}, type, \ldots)$; (2) *output actions*, which are messages of the form $(ID, \texttt{out}, type, \ldots)$; and (3) *protocol messages*, which are ordinary protocol messages to be delivered to other parties (of the form $(ID, type, \ldots)$). Note that input actions and output actions are local events within a party. Before a party starts to process messages that are tagged with ID,

[1]A more restrictive formalization of liveness conditions for environments with a computationally bounded scheduler is provided by Cachin et al. [5] through the notion of *probabilistically uniformly bounded statistics*; this notion can be easily applied to our protocol with some modifications, but we refrain from using it for the sake of readability.

the instance must be *initialized* by a special input action of the form $(ID, \texttt{in}, \texttt{open}, \textit{type})$, where *type* denotes the protocol type and/or its implementation. Such an action must precede any other input action with tag ID. We usually assume that it occurs implicitly with the first regular input action.

To analyze a protocol, we use two measures, message complexity and communication complexity. The message complexity of a protocol instance with tag ID is defined as the total number of all protocol messages with the tag ID or any tag starting with $ID|\ldots$ that correct parties generate. The communication complexity of a protocol instance with tag ID is defined as the total bit length of all protocol messages with the tag ID or any tag starting with $ID|\ldots$ that correct parties generate.

We make use of a digital signature scheme for our protocol. A digital signature scheme consists of algorithms for key generation, signing, and verification. As part of the system initialization, the dealer generates (using the key generation algorithm) the public key/private key pair for each party and gives every party its private key and the public keys of all parties. We assume that the signature scheme is secure in the sense of the standard security notion for signature schemes of modern cryptography, i.e., preventing existential forgery under adaptive chosen-message attacks [12].

Since we use the formal model of modern cryptography [11], we allow for a negligible probability of failure in the specification of our protocols.

2.2 Protocol Primitives

Our atomic broadcast protocol relies on a consistent broadcast protocol with special properties and on a Byzantine agreement protocol.

2.2.1 Strong Consistent Broadcast

We enhance the notion of consistent broadcast found in the literature [5] to develop the notion that we call *strong consistent broadcast*. Ordinary consistent broadcast provides a way for a designated sender P_s to broadcast a payload to all parties and requires that any two correct parties that deliver the payload agree on its content.

The standard protocol for implementing ordinary consistent broadcast is Reiter's *echo broadcast* [20]; it involves $\mathcal{O}(n)$ messages, has a latency of three message flows, and relies on a digital signature scheme. The sender starts the protocol by sending the payload m to all parties; then it waits for a quorum of $\lceil \frac{n+t+1}{2} \rceil$ parties to issue a signature on the payload and to "echo" the payload and the signature to the sender. When the sender has collected and verified enough signatures, it composes a final protocol message containing the signatures and sends it to all parties.

With a faulty sender, an ordinary consistent broadcast protocol permits executions in which some parties fail to deliver the payload when others succeed. Therefore, a useful enhancement of consistent broadcast is a transfer mechanism, which allows any party that has delivered the payload to help others do the same.

For reasons that will be evident later, we introduce another enhancement and require that when a correct party terminates a consistent broadcast and delivers a payload, there must be a quorum of at least $n - t$ parties (instead of only $\lceil \frac{n+t+1}{2} \rceil$) who participated in the protocol and approved the delivered payload. We call consistent broadcast with such a transfer mechanism and the special quorum rule *strong consistent broadcast*.

Formally, every broadcast instance is identified by a tag ID. At the sender P_s, strong consistent broadcast is invoked by an input action of the form $(ID, \texttt{in}, \texttt{sc-broadcast}, m)$, with $m \in \{0,1\}^*$. When that occurs, we say P_s *sc-broadcasts* m *with tag* ID. Only P_s executes this action; all other parties start the protocol only when they initialize instance ID in their role as receivers. A party terminates a consistent broadcast of m tagged with ID by generating an output action of the form $(ID, \texttt{out}, \texttt{sc-deliver}, m)$. In that case, we say P_i *sc-delivers* m *with tag* ID.

For the transfer mechanism, a correct party that has *sc-delivered* m with tag ID should be able to output a bit string M_{ID} that *completes* the *sc-broadcast* in the following sense: any correct party that has not yet *sc-delivered* m can run a *validation algorithm* on M_{ID} (this may involve a public key associated with the protocol), and if M_{ID} is determined to be *valid*, it can also *sc-deliver* m from M_{ID}.

Definition 1 (Strong consistent broadcast). A protocol for strong consistent broadcast satisfies the following conditions except with negligible probability.

Termination: If a correct party *sc-broadcasts* m with tag ID, then all correct parties eventually *sc-deliver* m with tag ID.

Agreement: If two correct parties P_i and P_j *sc-deliver* m and m' with tag ID, respectively, then $m = m'$.

Integrity: Every correct party *sc-delivers* at most one payload m with tag ID. Moreover, if the sender P_s is correct, then m was previously *sc-broadcast* by P_s with tag ID.

Transferability: After a correct party has *sc-delivered* m with tag ID, it can generate a string M_{ID} such that any correct party that has not *sc-delivered* a message with tag $ID$is able to *sc-deliver* some message immediately upon processing M_{ID}.

Strong unforgeability: For any ID, it is computationally infeasible to generate a value M that is accepted as valid by the validation algorithm for completing ID unless $n - 2t$ correct parties have initialized instance ID and actively participated in the protocol.

Note that the termination, agreement, and integrity properties are the same as in ordinary consistent broadcast [20][5].

Given the above implementation of consistent broadcast, one can obtain strong consistent broadcast with two simple modifications. The completing string M_{ID} for ensuring transferability consists of the final protocol message; the attached signatures are sufficient for any other party to complete the *sc-broadcast*. Strong unforgeability is obtained by setting the signature quorum to $n - t$.

With signatures of size K bits, the echo broadcast protocol has communication complexity $\mathcal{O}\big(n(|m|+nK)\big)$ bits, where $|m|$ denotes the bit length of the payload m. By replacing the quorum of signatures with a threshold signature [9], it is possible to reduce the communication complexity to $\mathcal{O}\big(n(|m|+K)\big)$ bits [5], under the reasonable assumption that the lengths of a threshold signature and a signature share are also at most K bits [24].

In the rest of the paper, we assume that strong consistent broadcast is implemented by applying the above modifications to the echo broadcast protocol with threshold signatures. Hence, the length of a completing string is $\mathcal{O}(|m| + K)$ bits.

2.2.2 Multi-Valued Byzantine Agreement

We use a protocol for multi-valued Byzantine agreement (MVBA) as defined by Cachin et al. [5], which allows agreement values from an arbitrary domain instead of being restricted to binary values. Unlike previous multi-valued Byzantine agreement protocols, their protocol does not allow the decision to fall back on a *default* value if not all correct parties propose the same value, but uses a protocol-external mechanism instead. This so-called *external validity condition* is specified by a global, polynomial-time computable predicate Q_{ID}, which is known to all parties and is typically determined by an external application or higher-level protocol. Each party proposes a value that contains certain validation information. The protocol ensures that the decision value was proposed by at least one party, and that the decision value satisfies Q_{ID}.

When a party P_i starts an MVBA protocol instance with tag ID and an input value $v \in \{0,1\}^*$ satisfying predicate Q_{ID}, we say that P_i *proposes* v *for multi-valued agreement with tag ID and predicate* Q_{ID}. Correct parties only propose values that satisfy Q_{ID}. When P_i terminates the MVBA protocol instance with tag ID and outputs a value v, we say that it *decides* v *for* ID.

Definition 2 (Multi-valued Byzantine agreement). A protocol for *multi-valued Byzantine agreement* with predicate Q_{ID} satisfies the following conditions except with negligible probability.

External Validity: Any correct party that decides for ID decides v such that $Q_{ID}(v)$ holds.
Agreement: If some correct party decides v for ID, then any correct party that decides for ID decides v.
Integrity: If all parties are correct and if some party decides v for ID, then some party proposed v for ID.
Termination: All correct parties eventually decide for ID.

The MVBA protocol of Cachin et al. [5] builds upon a protocol for binary Byzantine agreement (such as the one of Cachin et al. [6]), which relies on threshold signatures and a threshold coin-tossing protocol (e.g., [6]). The expected message complexity of the MVBA protocol is $\mathcal{O}(n^2)$ and the expected communication complexity is $\mathcal{O}\big(n^3 + n^2(K + L)\big)$, where K is the length of a threshold signature and L is a bound on the length of the values that can be proposed.

2.3 Definition of Atomic Broadcast

Atomic broadcast provides a "broadcast channel" abstraction [13], such that all correct parties deliver the same set of messages broadcast on the channel in the same order. A party P_i *atomically broadcasts* (or *a-broadcasts*) a payload m with tag ID when an input action of the form $(ID, \texttt{in}, \texttt{a-broadcast}, m)$ with $m \in \{0,1\}^*$ is delivered to P_i. Broadcasts are parameterized by the tag ID to identify their corresponding broadcast channel. A party *atomically delivers* (or *a-delivers*) a payload m with tag ID by generating an output action of the form $(ID, \texttt{out}, \texttt{a-deliver}, m)$. A party may *a-broadcast* and *a-deliver* an arbitrary number of messages with the same tag.

Definition 3 (Atomic broadcast). A protocol for atomic broadcast satisfies the following properties except with negligible probability.

Validity: If $t + 1$ correct parties *a-broadcast* some payload m with tag ID, then some correct party eventually *a-delivers* m with tag ID.
Agreement: If some correct party has *a-delivered* m with tag ID, then all correct parties eventually *a-deliver* m with tag ID.
Total Order: If two correct parties both *a-delivered* distinct payloads m_1 and m_2 with tag ID, then they have *a-delivered* them in the same order.
Integrity: For any payload m, a correct party P_j *a-delivers* m with tag ID at most once. Moreover, if all parties are correct, then m was previously *a-broadcast* by some party with tag ID.

The above properties are similar to the definitions of Cachin et al. [5] and of Kursawe and Shoup [15]. We do not formalize their *fairness* condition, which requires that the protocol "makes progress" towards delivering a payload as soon as $t+1$ correct parties have *a-broadcast* it. However, our protocol actually satisfies an equivalent notion (cf., Lemma 4).

3 The Parsimonious Asynchronous Atomic Broadcast Protocol

3.1 Overview

The starting point for the development of our Protocol PABC is the BFT protocol [8], which can be seen as the adaptation of Lamport's Paxos consensus protocol [16] to tolerate Byzantine faults. In the BFT protocol, a leader determines the delivery order of payloads and conveys the order using reliable broadcast to other parties. The parties then atomically deliver the payloads in the order chosen by the leader. If the leader appears to be slow or exhibits faulty behavior, a party switches to the recovery mode. When enough correct parties have switched to recovery mode, the protocol ensures that all correct parties eventually start the recovery phase. The goal of the recovery phase is to start the next epoch in a consistent state and with a new leader. The difficulty lies in determining which payloads have been delivered in the optimistic phase of the past epoch. The BFT protocol delegates this task to the leader of the new epoch. But since the recovery phase of BFT is also deterministic, it may be that the new leader is evicted immediately, before it can do any useful work, and the epoch passes without delivering any payloads. This denial-of-service attack against the BFT protocol violates liveness but is unavoidable in asynchronous networks.

The KS protocol [15] prevents this attack by ensuring that at least one payload is delivered during the recovery phase. It employs a round of randomized Byzantine agreement to agree on a set of payloads for atomic delivery, much like the asynchronous atomic broadcast protocol of Cachin et al. [5]. During the optimistic phase, the epoch leader conveys the delivery order through reliable broadcast as in BFT, which leads to an amortized message complexity of $\mathcal{O}(n^2)$.

Our approach is to replace reliable broadcast in the KS protocol with strong consistent broadcast; the replacement directly leads to an amortized message complexity of only $\mathcal{O}(n)$. But the replacement also introduces complications in the recovery phase, since a corrupted leader may cause the fate of some payloads to be undefined in the sense that there might be only a single correct party that has *sc-delivered* a payload, but no way for other correct parties to learn about this fact. We solve this problem by delaying the atomic delivery of an *sc-delivered* payload until more payloads have been *sc-delivered*. However, the delay introduces an additional problem of payloads getting "stuck" if no further payloads arrive. We address this by having the leader generate *dummy* payloads when no further payloads arrive within a certain time window.

The recovery phase in our protocol has a structure similar to that of the KS protocol, but is simpler and more efficient. At a high level, a first MVBA instance ensures that all correct parties agree on a synchronization point. Then, the protocol ensures that all correct parties *a-deliver* the payloads up to that point; to implement this step, every party must store all payloads that were delivered in the optimistic phase, together with information that proves the fact that they were delivered. A second MVBA instance is used to *a-deliver* at least one payload, which guarantees that the protocol makes progress in every epoch.

3.2 Details

We now describe the optimistic and the recovery phases in detail. The line numbers refer to the detailed protocol description in Figures 1–3.

3.2.1 Optimistic Phase

Every party keeps track of the current epoch number e and stores all payloads that it has received to *a-broadcast* but not yet *a-delivered* in its *initiation queue* $\mathcal{I}$. An element x can be appended to $\mathcal{I}$ by an operation $append(x, \mathcal{I})$, and an element x that occurs anywhere in $\mathcal{I}$ can be removed by an operation $remove(x, \mathcal{I})$. A party also maintains an array log of size B that acts as a buffer for all payloads to *a-deliver* in the current epoch. Additionally, a party stores a set $\mathcal{D}$ of all payloads that have been *a-delivered* so far.

We describe the optimistic phase of Protocol PABC by first detailing the normal protocol operation when the leader functions properly, and then explaining the mechanisms that ensure that the protocol switches to the recovery phase when the leader is not functioning properly.

Normal Protocol Operation. When a party receives a request to *a-broadcast* a payload m, it appends m to $\mathcal{I}$ and immediately forwards m using an `initiate` message to the leader P_l of the epoch, where $l = e \mod n$ (lines 12–14). When this happens, we say P_i *initiates* the payload.

The leader binds sequence numbers to the payloads that it receives in `initiate` messages, and conveys the bindings to the other parties through strong consistent broadcast. For this purpose, all parties execute a loop (lines 15–38) that starts with an instance of strong consistent broadcast (lines 15–26). The leader acts as the sender of strong consistent broadcast and the tag contains the epoch e and a sequence number s. Here, s starts from 0 in every epoch. The leader *sc-broadcasts* the next available initiated

Protocol PABC **for party** P_i **and tag** ID

initialization:

```
 1: $e \leftarrow 0$                                   {current epoch}
 2: $\mathcal{I} \leftarrow []$                        {initiation queue, list of *a-broadcast* but not *a-delivered* payloads}
 3: $\mathcal{D} \leftarrow \emptyset$                 {set of *a-delivered* payloads}
 4: $init\_epoch()$
```

function $init\_epoch()$**:**

```
 5: $l \leftarrow (e \bmod n) + 1$                     {$P_l$ is leader of epoch $e$}
 6: $log \leftarrow []$                                {array of size $B$ containing payloads committed in current epoch}
 7: $s \leftarrow 0$                                   {sequence number of next payload within epoch}
 8: $complained \leftarrow \texttt{false}$             {indicates if this party already complained about $P_l$}
 9: $start\_recovery \leftarrow \texttt{false}$        {signals the switch to the recovery phase}
10: $c \leftarrow 0$                                   {number of complain messages received for epoch leader}
11: $\mathcal{S} \leftarrow \mathcal{D}$               {set of *a-delivered* or already *sc-broadcast* payloads at $P_l$}
```

upon $(ID, \texttt{in}, \texttt{a-broadcast}, m)$:

```
12: send $(ID, \texttt{initiate}, e, m)$ to $P_l$
13: $append(m, \mathcal{I})$
14: $update_{\mathcal{F}_l}(\texttt{initiate}, m)$
```

forever: {optimistic phase}

```
15: if $\neg complained$ then {leader $P_l$ is not suspected}
16:     initialize an instance of strong consistent broadcast with tag $ID|\texttt{bind}.e.s$
17: $m \leftarrow \perp$
18: if $i = l$ then
19:     wait for $timeout(T)$ or receipt of a message $(ID, \texttt{initiate}, e, m)$ such that $m \notin \mathcal{S}$
20:     if $timeout(T)$ then
21:         $m \leftarrow \texttt{dummy}$
22:     else
23:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{m\}$
24:         $stop(T)$
25:     *sc-broadcast* the message $m$ with tag $ID|\texttt{bind}.e.s$
26: wait for $start\_recovery$ or *sc-delivery* of some $m$ with tag $ID|\texttt{bind}.e.s$ such that $m \notin \mathcal{D} \cup log$
27: if $start\_recovery$ then
28:     $recovery()$
29: else
30:     $log[s] \leftarrow m$
31:     if $s \geq 2$ then
32:         $update_{\mathcal{F}_l}(\texttt{deliver}, log[s-2])$
33:         $deliver(log[s-2])$
34:     if $i = l$ and $(log[s] \neq \texttt{dummy}$ or $(s > 0$ and $log[s-1] \neq \texttt{dummy}))$ then
35:         $start(T)$
36:     $s \leftarrow s + 1$
37:     if $s \bmod B = 0$ then
38:         $recovery()$
```

function $deliver(m)$**:**

```
39: if $m \neq \texttt{dummy}$ then
40:     $remove(m, \mathcal{I})$
41:     $\mathcal{D} \leftarrow \mathcal{D} \cup \{m\}$
42:     output $(ID, \texttt{out}, \texttt{a-deliver}, m)$
```

Figure 1: Protocol PABC for Atomic Broadcast (Part I)

```
function $complain()$:
    43: send $(ID, \texttt{complain}, e)$ to all parties
    44: $complained \leftarrow \texttt{true}$

upon receiving message $(ID, \texttt{complain}, e)$ from $P_j$ for the first time:
    45: $c \leftarrow c + 1$
    46: if $(c = t + 1)$ and $\neg complained$ then
    47:     $complain()$
    48: else if $c = 2t + 1$ then
    49:     $start\_recovery \leftarrow \texttt{true}$

let $Q_{ID|\texttt{watermark}.e}$ be the following predicate:
    $Q_{ID|\texttt{watermark}.e}\Big([(s_1, C_1, \sigma_1), \ldots, (s_n, C_n, \sigma_n)]\Big) \equiv$
      $\big($for at least $n - t$ distinct $j$, $s_j \neq \bot\big)$ and
        $\big($for all $j = 1, \ldots, n$, either $s_j = \bot$ or
          $(\sigma_j$ is a valid signature by $P_j$ on $(ID, \texttt{committed}, e, s_j, C_j)$ and
            $(s_j = -1$ or the value $C_j$ completes the sc-broadcast with tag $ID|\texttt{bind}.e.s_j)))\big)$

Let $Q_{ID|\texttt{deliver}.e}$ be the following predicate:
    $Q_{ID|\texttt{deliver}.e}\Big([(\mathcal{I}_1, \sigma_1), \ldots, \mathcal{I}_n, \sigma_n)]\Big) \equiv$ for at least $n - t$ distinct $j$,
        $\big(\mathcal{I}_j \cap \mathcal{D} = \emptyset$ and $\sigma_j$ is a valid signature by $P_j$ on $(ID, \texttt{queue}, e, j, \mathcal{I}_j)\big)$
```

Figure 2: Protocol PABC for Atomic Broadcast (Part II)

payload, and every party waits to *sc-deliver* some payload m. When m is *sc-delivered*, P_i stores it in *log*, but does not yet *a-deliver* it (line 30). At this point in time, we say that P_i has *committed* sequence number s to payload m in epoch e. Then, P_i *a-delivers* the payload to which it has committed the sequence number $s - 2$ (if available, lines 31–33). It increments s (line 36) and returns to the start of the loop.

Delaying the *a-delivery* of the payload committed to s until sequence number $s + 2$ has been committed is necessary to prevent the above problem of payloads whose fate is undefined. However, the delay results in another problem if no further payloads, those with sequence numbers higher than s, are *sc-delivered*. We solve this problem by instructing the leader to send `dummy` messages to eject the original payload(s) from the buffer. The leader triggers such a `dummy` message whenever a corresponding timer T expires (lines 20–21); T is activated whenever one of the current or the preceding sequence numbers was committed to a non-`dummy` payload (lines 34–35), and T is disabled when the leader *sc-broadcasts* a non-`dummy` payload (line 24). Thus, the leader sends at most two `dummy` payloads to eject a non-`dummy` payload.

Failure Detection and Switching to the Recovery Phase. There are two conditions under which the protocol switches to recovery phase: (1) when B payloads have been committed (line 38) and (2) when the leader is not functioning properly. The first condition is needed to keep the buffer *log* bounded and the second condition is needed to prevent a corrupted leader from violating liveness.

To determine if the leader of the epoch performs its job correctly, every party has access to a leader failure detector $\mathcal{F}_l$. For simplicity, Figures 1–3 do not include the pseudocode for $\mathcal{F}_l$. The protocol provides an interface $complain()$, which $\mathcal{F}_l$ can asynchronously invoke to notify the protocol about its suspicion that the leader is corrupted. Our protocol synchronously invokes an interface $update_{\mathcal{F}_l}$ of $\mathcal{F}_l$ to convey protocol-specific information (during execution of the $update_{\mathcal{F}_l}$ call, $\mathcal{F}_l$ has access to all variables of Protocol PABC).

```
**function** $recovery()$:
    {*Part 1: agree on watermark*}
    50: compute a signature $\sigma$ on $(ID, \texttt{committed}, e, s-1)$
    51: send the message $(ID, \texttt{committed}, e, s-1, C, \sigma)$ to all parties, where $C$ denotes
          the bit string that completes the *sc-broadcast* with tag $ID|\texttt{bind}.e.(s-1)$
    52: $(s_j, C_j, \sigma_j) \leftarrow (\bot, \bot, \bot) \qquad (1 \le j \le n)$
    53: **wait for** $n-t$ messages $(ID, \texttt{committed}, e, s_j, C_j, \sigma_j)$ from distinct $P_j$ such that $C_j$ completes
          the *sc-broadcast* instance $ID|\texttt{bind}.e.s_j$ and $\sigma_j$ is a valid signature on $(ID, \texttt{committed}, e, s_j)$
    54: $W \leftarrow [(s_1, C_1, \sigma_1), \ldots, (s_n, C_n, \sigma_n)]$
    55: propose $W$ for MVBA with tag $ID|\texttt{watermark}.e$ and predicate $Q_{ID|\texttt{watermark}.e}$
    56: **wait for** MVBA with tag $ID|\texttt{watermark}.e$ to decide some $\bar{W} = [(\bar{s}_1, \bar{C}_1, \bar{\sigma}_1), \ldots, (\bar{s}_n, \bar{C}_n, \bar{\sigma}_n)]$
    57: $w \leftarrow \max\{\bar{s}_1, \ldots, \bar{s}_n\} - 1$

    {*Part 2: synchronize up to watermark*}
    58: $s' \leftarrow s - 2$
    59: **while** $s' \le \min\{s-1, w\}$ **do**
    60:     **if** $s' \ge 0$ **then**
    61:         $deliver(log[s'])$
    62:     $s' \leftarrow s' + 1$
    63: **if** $s > w$ **then**
    64:     **for** $j = 1, \ldots, n$ **do**
    65:         $u \leftarrow \max\{s_j, \bar{s}_j\}$
    66:         $\mathcal{M} \leftarrow \{M_v\}$ for $v = u, \ldots, w$, where $M_v$ completes the *sc-broadcast* instance $ID|\texttt{bind}.e.v$
    67:         send message $(ID, \texttt{complete}, \mathcal{M})$ to $P_j$
    68: **while** $s \le w$ **do**
    69:     **wait for** a message $(ID, \texttt{complete}, \bar{\mathcal{M}})$ such that $\bar{M}_s \in \bar{\mathcal{M}}$ completes *sc-broadcast*
              with tag $ID|\texttt{bind}.e.s$
    70:     use $\bar{M}_s$ to *sc-deliver* some $m$ with tag $ID|\texttt{bind}.e.s$
    71:     $deliver(m)$
    72:     $s \leftarrow s + 1$

    {*Part 3: deliver some messages*}
    73: compute a digital signature $\sigma$ on $(ID, \texttt{queue}, e, i, \mathcal{I})$
    74: send the message $(ID, \texttt{queue}, e, i, \mathcal{I}, \sigma)$ to all parties
    75: $(\mathcal{I}_j, \sigma_j) \leftarrow (\bot, \bot) \qquad (1 \le j \le n)$
    76: **wait for** $n-t$ messages $(ID, \texttt{queue}, e, j, \mathcal{I}_j, \sigma_j)$ from distinct $P_j$ such that
          $\sigma_j$ is a valid signature from $P_j$ and $\mathcal{I}_j \cap \mathcal{D} = \emptyset$
    77: $Q \leftarrow [(\mathcal{I}_1, \sigma_1), \ldots, (\mathcal{I}_n, \sigma_n)]$
    78: propose $Q$ for MVBA with tag $ID|\texttt{deliver}.e$ and predicate $Q_{ID|\texttt{deliver}.e}$
    79: **wait for** MVBA with tag $ID|\texttt{deliver}.e$ to decide some $\bar{Q} = [(\bar{\mathcal{I}}_1, \bar{\sigma}_1), \ldots, (\bar{\mathcal{I}}_n, \bar{\sigma}_n)]$
    80: **for** $m \in \bigcup_{j=1}^{n} \bar{\mathcal{I}}_j \setminus \mathcal{D}$, in some deterministic order **do**
    81:     $deliver(m)$
    82: $init\_epoch()$
    83: **for** $m \in \mathcal{I}$ **do**
    84:     send $(ID, \texttt{initiate}, e, m)$ to $P_l$
```

Figure 3: Protocol PABC for Atomic Broadcast (Part III)

An implementation of $\mathcal{F}_l$ can check whether the leader is making progress based on a timeout and protocol information as follows. Recall that every party maintains a queue $\mathcal{I}$ of initiated but not yet *a-delivered* payloads. When P_i has initiated some m, it calls $update_{\mathcal{F}_l}(\texttt{initiate}, m)$ (line 14); this starts a timer $T_{\mathcal{F}_l}$ unless it is already activated. When a payload is *a-delivered* during the optimistic phase, the call to $update_{\mathcal{F}_l}(\texttt{deliver}, m)$ (line 32) checks whether the *a-delivered* payload is the first undelivered payload in $\mathcal{I}$, and if it is, disables $T_{\mathcal{F}_l}$. When $T_{\mathcal{F}_l}$ expires, $\mathcal{F}_l$ invokes $complain()$.

When P_i executes $complain()$, it sends a `complain` message to all parties (line 43); it also sets the $complained$ flag (line 44) and stops participating in the *sc-broadcasts* by not initializing the next instance. When a correct party receives $2t+1$ `complain` messages, it enters the recovery phase. There is a complaint "amplification" mechanism by which a correct party that has received $t+1$ `complain` messages and has not yet complained itself joins the complaining parties by sending its own `complain` message. Complaint amplification ensures that when some correct party enters the recovery phase, all other correct parties eventually enter it as well.

3.2.2 Recovery Phase

The recovery phase consists of three parts: (1) determining a watermark sequence number, (2) synchronizing all parties up to the watermark, and (3) delivering some payloads before entering the next epoch.

Part 1: Agree on Watermark The first part of the recovery phase determines a *watermark* sequence number w with the properties that (a) at least $t+1$ correct parties have committed all sequence numbers less than or equal to w in epoch e, and (b) no sequence number higher than $w+2$ has been committed by a correct party in epoch e.

Upon entering the recovery phase of epoch e, a party sends out a signed `committed` message containing $s-1$, the highest sequence number that it has committed in this epoch. It justifies $s-1$ by adding the bit string C that completes the *sc-broadcast* instance with tag e and $s-1$ (lines 50–51). Then, a party receives $n-t$ such `committed` messages with valid signatures and valid completion bit strings. It collects the received `committed` messages in a *watermark proposal vector* W and proposes W for MVBA. Once the agreement protocol decides on a *watermark decision vector* $\bar{W}$ (lines 52–56), the watermark w is set to the maximum of the sequence numbers in $\bar{W}$ minus 1 (line 57).

Consider the maximal sequence number $\bar{s}_j$ in $\bar{W}$ and the corresponding $\bar{C}_j$. It may be that P_j is corrupted or that P_j is the only correct party that ever committed $\bar{s}_j$ in epoch e. But the values contain enough evidence to conclude that at least $n-2t \geq t+1$ correct parties contributed to this instance of strong consistent broadcast. Hence, these parties have previously committed $\bar{s}_j - 1$. This ensures the first property of the watermark above (see also Lemma 2).

Although one or more correct parties may have committed $w+1$ and $w+2$, none of them has already *a-delivered* the corresponding payloads, because this would contradict the definition of w. Hence, these sequence numbers can safely be discarded. The discarding also ensures the second property of the watermark above (see Lemma 5). It is precisely for this reason that we delay the *a-delivery* of a payload to which sequence number s was committed until $s+2$ has been committed. Without it, the protocol could end up in a situation where up to t correct parties *a-delivered* a payload with sequence number $w+1$ or $w+2$, but it would be impossible for all correct parties to learn about this fact and to learn the *a-delivered* payload.

Part 2: Synchronize up to Watermark The second part of the recovery phase (lines 58–72) ensures that all parties *a-deliver* the payloads with sequence numbers less than or equal to w. It does so in a straightforward way using the *transferability* property of strong consistent broadcast.

In particular, every correct party P_i that has committed sequence number w (there must be at least $t+1$ such correct parties by the definition of w) computes *completing strings* M_s for $s = 0, \ldots, w$

that complete the *sc-broadcast* instance with sequence number s. It can do so using the information stored in log. Potentially, P_i has to send $M_0, \ldots, M_w$ to *all* parties, but one can apply the following optimization to reduce the communication. Note that P_i knows from at least $n - t$ parties P_j their highest committed sequence number s_j (either directly from a `committed` message or from the watermark decision vector); if P_i knows nothing from some P_j, it has to assume $s_j = 0$. Then P_i simply sends a `complete` message with $M_{s_j+1}, \ldots, M_w$ to P_j for $j = 1, \ldots, n$. Every party receives these completing strings until it is able to *a-deliver* all payloads committed to the sequence numbers up to w.

Part 3: Deliver Some Messages Part 3 of the recovery phase (lines 73–84) ensures that the protocol makes progress by *a-delivering* some messages before the next epoch starts. In an asynchronous network, implementing this property must rely on randomized agreement or on a failure detector [10]. This part uses one round of MVBA and is derived from the atomic broadcast protocol of Cachin et al. [5].

Every party P_i sends a signed `queue` message with all undelivered payloads in its initiation queue to all others (lines 73–74), collects a vector Q of $n - t$ such messages with valid signatures (lines 75–77), and proposes Q for MVBA. Once the agreement protocol has decided on a vector $\bar{Q}$ (lines 78–79), party P_i delivers the payloads in $\bar{Q}$ according to some deterministic order (lines 80–81).

Then P_i increments the epoch number and starts the next epoch by re-sending `initiate` messages for all remaining payloads in its initiation queue to the new leader (lines 82–84).

3.3 Optimizations

Both the BFT and KS protocols process multiple sequence numbers in parallel using a sliding window mechanism. For simplicity, our protocol description does not include this optimization and processes only the highest sequence number during every iteration of the loop in the optimistic phase. However, Protocol PABC can easily be adapted to process Ω payloads concurrently. In that case, up to Ω *sc-broadcast* instances are active in parallel, and the delay of two sequence numbers between *sc-delivery* and *a-delivery* of a payload is set to 2Ω. In part 1 of the recovery phase, the watermark is set to the maximum of the sequence numbers in the watermark decision vector minus Ω, instead of the maximum minus 1.

In our protocol description, the leader *sc-broadcasts* one initiated payload at a time. However, Protocol PABC can be modified to process a *batch* of payload messages at a time by committing sequence numbers to batches of payloads, as opposed to single payloads. The leader *sc-broadcasts* a batch of payloads in one instance, and all payloads in an *sc-delivered* batch are *a-delivered* in some deterministic order. This optimization has been shown to increase the throughput of the BFT protocol considerably [8].

Although the leader failure detector described in Section 3.2.1 is sufficient to ensure liveness, it is possible to enhance it using protocol information as follows. The leader in the optimistic phase will never have to *sc-broadcast* more than two `dummy` messages consecutively to evict non-`dummy` payloads from the buffer. The failure detector oracle can maintain a counter to keep track of and restrict the number of successive `dummy` payloads *sc-broadcast* by the leader. If m is a non-`dummy` payload, the call to $update_{\mathcal{F}_l}(\texttt{deliver}, m)$ upon *a-delivery* of payload m resets the counter; otherwise, the counter is incremented. If the counter ever exceeds 2, then $\mathcal{F}_l$ invokes the $complain()$ function.

3.4 Protocol Complexity

In this section, we examine the message and communication complexities of our protocol. We assume that strong consistent broadcast is implemented by the echo broadcast protocol using threshold signatures, and that MVBA is implemented by the protocol of Cachin et al. [5], as described in Section 2.2.

For a payload m that is *a-delivered* in the optimistic phase, the message complexity is $\mathcal{O}(n)$, and the communication complexity is $\mathcal{O}\big(n(|m| + K)\big)$, where the length of a threshold signature and a signature share are at most K bits.

The recovery phase incurs higher message and communication complexities because it involves Byzantine agreement. The MVBA protocol of Cachin et al. [5] has an expected message complexity of $\mathcal{O}(n^2)$. Hence, determining the watermark in part 1 of the recovery involves expected $\mathcal{O}(n^2)$ messages. The corresponding expected communication complexity is $\mathcal{O}\big(n^3(|m| + K)\big)$ since the proposal values contain $\mathcal{O}(n)$ 3-tuples of the form (s_j, C_j, σ_j), each of length $\mathcal{O}(|m| + K)$. Here, m denotes the longest payload contained in the proposal.

In part 2 of the recovery phase, up to $\mathcal{O}(n^2)$ `complete` messages are exchanged. Recall that a `complete` message may encompass all payload messages that were previously *a-delivered* in the optimistic phase of the epoch. Each of the $w \leq B$ completing strings in a `complete` message may be $\mathcal{O}(|m| + K)$ bits long, where m denotes the longest *a-delivered* payload. Hence, the communication complexity of part 2 of the recovery phase is $\mathcal{O}\big(n^2B(|m| + K)\big)$.

Part 3 of the recovery phase is again dominated by the cost of the MVBA protocol. Hence, the expected message complexity of part 3 is $\mathcal{O}(n^2)$ and the expected communication complexity is $\mathcal{O}(n^3|m|)$ since the proposal values in MVBA are of length $\mathcal{O}(n|m|)$.

To summarize, for a payload that is *a-delivered* in the recovery phase, the cost is dominated by the MVBA protocol, resulting in an expected message complexity of $\mathcal{O}(n^2)$ and an expected communication complexity of $\mathcal{O}\big(n^2(n + B)(|m| + K)\big)$. Assuming that the protocol stays in the optimistic mode as long as possible and *a-delivers* B payloads before executing recovery, the *amortized* expected complexities per payload over an epoch are $\mathcal{O}(n + \frac{n^2}{B})$ messages and $\mathcal{O}\big(\frac{n^3}{B}(|m| + K)\big)$ bits. It is reasonable to set $B \gg n$, so that we achieve amortized expected message complexity $\mathcal{O}(n)$ as claimed.

4 Analysis

In this section, we prove the following theorem.

Theorem 1. *Given a digital signature scheme, a protocol for strong consistent broadcast, and a protocol for multi-valued Byzantine agreement, Protocol* PABC *provides atomic broadcast for* $n > 3t$.

We first establish some technical lemmas that describe the properties of Protocol PABC.

Lemma 2. *At the point in time when the first correct party has determined the watermark* w *during the recovery phase of epoch* e*, at least* $t + 1$ *correct parties have committed sequence number* w *in epoch* e.

Proof. First note that the lemma holds trivially if $w = -2$, and we may assume $w \geq -1$ in the rest of the proof. Let j^* denote the index of the largest sequence number $\bar{s}_1, \ldots, \bar{s}_n$ contained in the decision vector $\bar{W}$ of the agreement with tag $ID|\texttt{watermark}.e$. Note that $w = \bar{s}_{j^*} - 1$ according to the protocol. By the predicate $Q_{ID|\texttt{watermark}.e}$, the string $\bar{C}_{j^*}$ in $\bar{W}$ completes the strong consistent broadcast with tag $ID|\texttt{bind}.e.j^*$. According to the strong unforgeability property of strong consistent broadcast, $\bar{C}_{j^*}$ contains evidence that at least $n - 2t$ distinct correct parties have participated in the *sc-broadcast* instance with sequence number j^*. According to the logic of the optimistic phase, a correct party initializes an instance of strong consistent broadcast with tag $ID|\texttt{bind}.e.s$ only after committing sequence number $s - 1$. Hence, these $n - 2t \geq t + 1$ correct parties have also committed sequence number $j^* - 1 = w$. □

Lemma 3. *If some correct party has entered the recovery phase of epoch* e*, then all correct parties eventually enter epoch* $e + 1$.

Proof. To establish the above lemma, we prove the following two claims.

Claim 1: If some correct party has entered the recovery phase of epoch e, then all correct parties eventually enter the recovery phase of epoch e.

Claim 2: If all correct parties have entered the recovery phase of epoch e, then all correct parties eventually enter epoch $e+1$.

By the transitive application of the two claims, the lemma follows.

We first prove Claim 1. Suppose that a correct party P_i enters the recovery phase of epoch e. P_i does so only after receiving `complain` messages from $2t+1$ distinct parties. At least $t+1$ of these messages must have been from correct parties. Hence, every correct party eventually receives $t+1$ `complain` messages and sends its own `complain` message. Thus, every correct party eventually receives $n-t \geq 2t+1$ `complain` messages and transitions to the recovery phase of epoch e.

To prove Claim 2, one has to show that a correct party that enters the recovery phase of epoch e eventually completes all three parts of the recovery and moves to epoch $e+1$.

A correct party completes part 1 of the recovery because it eventually receives $n-t$ valid `committed` messages from all correct parties and because all correct parties eventually decide in the MVBA protocol, according to its termination property.

Part 2 of the recovery phase is concerned with ensuring that all correct parties *a-deliver* the set of non-`dummy` payloads to which sequence numbers less than or equal to w were committed. Completion of part 2 by a correct party is guaranteed by the transferability property of strong consistent broadcast as follows. A correct party P_i that has committed sequence number w first *a-delivers* non-`dummy` payloads committed to sequence numbers $w-1$ and w (if it has not already done so). Then it sends a message with a set of completing strings $\{M_s | \, 0 \leq s \leq w\}$ to all other parties and moves to part 3 of the recovery phase. Here, M_s is the string that completes the *sc-broadcast* instance with sequence number s, which can be computed from the information stored in log. A correct party P_j that has not committed all sequence numbers less than w waits to receive the corresponding completing strings; P_j is guaranteed to receive them eventually, since, by Lemma 2, there are at least $t+1$ correct parties that have committed sequence number w. P_j then *a-delivers* all non-`dummy` payloads with sequence numbers up to w and moves to part 3 of the recovery phase.

Analogous to part 1, completion of part 3 of the recovery is guaranteed by the fact that $n-t$ `queue` messages will eventually be received and by the termination property of MVBA. □

Lemma 4. *Suppose e^* is the largest epoch number at any correct party at the point in time when $t+1$ correct parties have* a-broadcast *some payload m, and assume that some correct party did not* a-deliver *m before entering epoch e^*. Then some correct party P_i* a-delivers *m before entering epoch e^*+1.*

Proof. The lemma is trivially satisfied if P_i *a-delivers* m during the optimistic phase of epoch e^*. Otherwise, m is still present in the initiation queue $\mathcal{I}$ of at least $t+1$ correct parties. Since the initiation queues of $n-t$ parties are included in the decision vector of MVBA in part 3 of the recovery phase, at least one of these queues also contains m, and the lemma follows. □

Lemma 5. *Let w be the watermark of epoch e. No correct party commits a sequence number larger than $w+2$ in epoch e, and no correct party* a-delivers *a payload to which a sequence number larger than w has been committed in epoch e before reaching part 3 of the recovery phase.*

Proof. The proof is by contradiction. Suppose that some correct party P_i has committed sequence number $w' = w+3$. Then, P_i has previously *sc-delivered* some m with tag $ID|\texttt{bind}.e.w'$, and the strong unforgeability property of strong consistent broadcast implies that at least $n-2t \geq t+1$ correct parties have participated in this *sc-broadcast* instance. Since correct parties initialize the *sc-broadcast* instance with tag $ID|\texttt{bind}.e.w'$ only after committing the previous sequence number, they have also committed sequence number $w'-1$.

Therefore, these $t+1$ correct parties have also sent a signed `committed` message containing sequence number $w'-1$ during recovery. Hence, the decision vector $\bar{W}$ with $n-t$ entries signed by distinct parties contained a triple $(\bar{s}_{j^*}, \bar{C}_{j^*}, \bar{\sigma}_{j^*})$ signed by one of those $t+1$ correct parties with $\bar{s}_{j^*} = w'-1$. By the agreement property of MVBA, every correct party must have computed the same

$\bar{W}$ and set w to the maximum among the $\bar{s}_j$ values contained in $\bar{W}$ minus 1, i.e. $w = \bar{s}_{j^*} - 1 = w' - 2$. But this contradicts our assumption that $w' = w + 3$.

To prove the second part of the lemma, recall that the *a-delivery* of the payload to which sequence number $s-2$ has been committed is delayed until after sequence number s has been committed. But since no correct party commits a sequence number larger than $w+2$, as shown in the first part of the lemma, no correct party *a-delivers* any payload to which a sequence number larger than w has been committed in the optimistic phase of epoch e. After the watermark in part 1 of the recovery phase has been determined, as can be seen from the checks in lines 59 and 68, part 2 ensures that payloads are *a-delivered* only up to the watermark; sequence numbers $w+1$ and $w+2$ are simply discarded. $\square$

Lemma 6. *Suppose the watermark of epoch e satisfies $w \geq -1$. Then all correct parties eventually* a-deliver *all non-*`dummy` *payloads to which any correct party has committed a sequence number less than or equal to w in epoch e.*

Proof. By the agreement and termination properties of MVBA, a correct party P_i eventually determines the watermark w of epoch e. During the optimistic phase, it has *a-delivered* all non-`dummy` payloads with sequence numbers less than $s-2$.

When P_i moves to part 2 of the recovery phase, the code in lines 59–62 ensures that P_i also *a-delivers* those non-`dummy` payloads to which sequence numbers $s-2$ and $s-1$ have been committed.

Note that w may be smaller or larger than $s-1$, the highest committed sequence number. If $w < s$ and P_i has already committed w, then by the logic of the optimistic phase and the loop in lines 59–62, P_i eventually *a-delivers* all non-`dummy` payloads to which a sequence number less than or equal to w has been committed.

On the other hand, if $w \geq s$ and P_i has not yet committed w, it waits to receive a string $\{M_{s'}\}$ that completes the *sc-broadcast* instance with sequence number s' for $s' \leq w$. Party P_i is guaranteed to receive all of them eventually, since there are at least $t+1$ correct parties that have committed all sequence numbers up to w by Lemma 2. P_j then *a-delivers* all non-`dummy` payloads to which sequence numbers between s and w have been committed in epoch e. $\square$

Lemma 7. *In every epoch e, there exists a sequence S of payloads such that any correct party* a-delivers *all payloads in epoch e in the order of S.*

Proof. We first define a sequence S_i' for every correct P_i and show that the sequences computed by distinct correct parties are equal.

S_i' is defined as follows. During the optimistic phase, all payloads that P_i *sc-delivers* are appended to S_i' in the order of their delivery. Suppose P_i has entered the recovery phase and has computed the watermark w; note that S_i' contains s elements. If $s > w+1$, then S_i' is truncated to the first $w+1$ elements; if $s \leq w$, then S_i' is extended to $w+1$ elements by the payloads that are *sc-delivered* with tags $ID|\texttt{bind}.e.v$ for $v = s, \ldots, w$ through the `complete` messages in lines 68–72. During part 3 of the recovery phase, all payloads in $\cup_{j=1}^{n} \bar{\mathcal{I}}_j \setminus \mathcal{D}$ are appended to S_i' according to the given deterministic order, according to line 80.

It is easy to see that, except with negligible probability, $S_i' = S_j'$ for any two correct parties P_i and P_j from the consistency property of strong consistent broadcast and from the agreement property of MVBA. Hence, one may think of a global sequence $S' = S_i'$ for all P_i.

Note that every party *a-delivers* all non-`dummy` payloads in S_i' during epoch e. Hence, the sequence S is equal to S' with all `dummy` payloads removed. $\square$

Proof of Theorem 1. We have to show that Protocol PABC satisfies the validity, agreement, total order, and integrity properties of atomic broadcast.

Validity follows directly from Lemma 4. In fact, Lemma 4 proves a stronger version of the validity property stated in Section 2.3. The reason is that while the validity property specifies only an "eventual *a-delivery*" for a payload m that has been *a-broadcast* by $t + 1$ correct parties, Lemma 4 shows that m will be delivered relatively quickly.

To show *Agreement*, suppose that a correct party P_i has *a-delivered* some m in epoch e. We have to show that eventually all correct parties *a-deliver* m.

We first distinguish two cases. In the first case, suppose P_i has *a-delivered* m before entering part 3 of the recovery phase in epoch e. Then, Lemma 5 proves that a sequence number less than or equal to the watermark w of epoch e has been committed to m. Lemma 6 shows that all correct parties eventually *a-deliver* all non-dummy payloads to which a sequence number less than or equal to w has been committed, including m. This proves that the agreement property holds for the first case.

In the second case, m was *a-delivered* during part 3 of the recovery phase, after P_i had terminated the MVBA protocol. Then the agreement and termination properties of MVBA guarantee that all correct parties eventually terminate the MVBA protocol and *a-deliver* the same sequence of payloads.

Hence, we have proved that if P_i *a-delivers* m in epoch e, all correct parties in epoch e eventually *a-deliver* m. Extending the proof to the definition of agreement now only requires us to show that all correct parties eventually reach epoch e. Lemma 3 implies exactly that. Hence, by induction on the epoch, it can be easily seen that Protocol PABC satisfies agreement.

The *total order* property for a particular epoch e is proved by Lemma 7. Hence, by induction on the epoch number, it can be easily seen that Protocol PABC satisfies total order.

For *integrity*, we first show that a payload m is *a-delivered* at most once by a correct party P_i. Suppose that P_i *a-delivers* m in epoch e. Then, there are two possibilities, depending on whether the *a-delivery* happened before or after part 3 of epoch e's recovery phase was entered. In the former case (*a-delivery* before part 3 is entered), some sequence number less than or equal to w must have been committed to m. The check in line 26 ensures that a sequence number is committed to payload m only if $m \notin \mathcal{D}$. In the latter case (*a-delivery* in part 3 of the recovery phase), payload m must have been a part of the decision vector $\bar{Q}$. The check in line 80 ensures that only payloads in $\bar{Q}$ that are not in $\mathcal{D}$ are *a-delivered* in a deterministic order. Hence, it is clear that a payload m is *a-delivered* at most once by P_i. Even if corrupted parties *a-broadcast* payloads that have already been *a-delivered*, they are not *a-delivered* again.

The second part of the integrity property, i.e., that our protocol only *a-delivers* payloads that were previously *a-broadcast* by some party if all parties are correct, is trivially satisfied by the protocol. □

5 Discussion

In this section, we discuss the practical significance of our optimistic protocol and compare it with other efficient atomic broadcast protocols.

5.1 Practical Significance

In our formal system model, the adversary controls the scheduling of messages and hence the time-outs; thus, the adversary can cause parties to complain about a correctly functioning leader resulting in unnecessary transitions from the optimistic phase to the recovery phase.

Unlike the adversary in our formal model, the network in a real-world setting will not always behave in the worst possible manner. The motivation for Protocol PABC — or any optimistic protocol such as the BFT and KS protocols for that matter — is the hope that timing assumptions based on stable network conditions have a high likelihood of being accurate. Such a hope is realistic since practical observations show that network behavior alternates between long periods of stable conditions and relatively short periods of instability; this indicates that unstable network conditions are the exception rather than the norm. During periods of stability and when no new intrusions are detected, the optimistic assumption will be satisfied and our protocol will make fast progress in the optimistic phase. However, both safety

Table 1: Comparison of Efficient Byzantine-Fault-Tolerant Atomic Broadcast Protocols

Protocol	Synchrony for Safety?	Synchrony for Liveness?	Public-key Operations?	Message Complexity	
				Normal Cond.	Worst-case
Rampart [21]	yes	yes	yes	$\mathcal{O}(n)$	unbounded
SecureRing [14]	yes	yes	yes	$\mathcal{O}(n)$	unbounded
ITUA [19]	yes	yes	yes	$\mathcal{O}(n)$	unbounded
Cachin et al. [5]	no	no	yes	expected $\mathcal{O}(n^2)$	expected $\mathcal{O}(n^2)$
BFT [8]	no	yes	no	$\mathcal{O}(n^2)$	unbounded
KS [15]	no	no	yes	$\mathcal{O}(n^2)$	expected $\mathcal{O}(n^2)$
Protocol PABC	no	no	yes	$\mathcal{O}(n)$	expected $\mathcal{O}(n^2)$

and liveness are still guaranteed even if the network is unstable, as long as no more than $t < n/3$ parties are actively misbehaving.

5.2 Comparison

Table 1 compares the synchrony assumptions, cryptographic requirements, and message complexity of Protocol PABC with the other recent Byzantine-fault-tolerant atomic broadcast protocols mentioned in the introduction. We devote the rest of this section to a more elaborate comparison with the two protocols closest to ours, namely the BFT protocol and the KS protocol.

Under stable network conditions and with a correct leader, all three protocols operate in their optimistic phases. These conditions are likely to apply during most of the running time of the system. In this case, the linear message and communication complexities of Protocol PABC compare favorably with the quadratic complexities of the BFT and KS protocols.

Under unstable network conditions, the deterministic BFT protocol can generate a potentially unbounded number of protocol messages by repeatedly switching from one epoch to another without making progress. This represents a violation of liveness and is prevented in the KS protocol and in Protocol PABC, since their recovery phases rely on randomized agreement and *a-deliver* some payloads. Naturally, using Byzantine agreement makes our recovery phase more expensive than the one of the BFT protocol.

The recovery phase of Protocol PABC is slightly more efficient than that of the KS protocol. The KS protocol requires four iterations of Byzantine agreement in addition to one iteration for each concurrently handled reliable broadcast instance. The recovery phase of our protocol uses only two iterations of Byzantine agreement, irrespective of the number of strong consistent broadcast instances that are concurrently handled.

6 Conclusion

We described a protocol that, for the first time, achieves asynchronous atomic broadcast with $\mathcal{O}(n)$ amortized expected messages per payload message. The previous best solutions used $\Theta(n^2)$ messages. Despite intrusions and instability, our protocol guarantees both safety and liveness as long as no more than $t < n/3$ parties are corrupted by the adversary. Our use of strong consistent broadcast, instead of reliable broadcast as in the BFT and KS protocols, introduces an additional digital signature computation at each party for every delivered payload. However, the intended deployment environments for our protocol are WANs, where message latency typically exceeds the time to perform digital signature computations; hence, we expect our protocol to be significantly more efficient than previous protocols in this case.

Acknowledgments

This work was supported in part by NSF under Grant No. CNS-0406351. We are grateful to Bill Sanders for support, interesting discussions, and comments on improving the quality of the paper. We also thank Jenny Applequist for her editorial comments.

References

[1] P. Berman and A. A. Bharali, "Quick Atomic Broadcast," in *Proc. 7th International Workshop on Distributed Algorithms (WDAG)*, vol. 725 of *Lecture Notes in Computer Science*, pp. 189–203, Springer, 1993.

[2] P. Berman and J. A. Garay, "Randomized Distributed Agreement Revisited," in *Proc. 23th International Symposium on Fault-Tolerant Computing (FTCS-23)*, pp. 412–419, 1993.

[3] G. Bracha, "An Asynchronous $[(n-1)/3]$-Resilient Consensus Protocol," in *Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 154–162, 1984.

[4] C. Cachin, "Distributing Trust on the Internet," in *Proc. International Conference on Dependable Systems and Networks (DSN-2001)*, pp. 183–192, June 2001.

[5] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and Efficient Asynchronous Broadcast Protocols (Extended Abstract)," in *Advances in Cryptology: CRYPTO 2001* (J. Kilian, ed.), vol. 2139 of *Lecture Notes in Computer Science*, pp. 524–541, Springer, 2001.

[6] C. Cachin, K. Kursawe, and V. Shoup, "Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement using Cryptography," *Journal of Cryptology*, vol. 18, no. 3, 2005.

[7] R. Canetti and T. Rabin, "Fast Asynchronous Byzantine Agreement with Optimal Resilience," in *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 42–51, 1993.

[8] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, pp. 398–461, Nov. 2002.

[9] Y. Desmedt, "Society and Group Oriented Cryptography: A New Concept," in *Advances in Cryptology: CRYPTO '87* (C. Pomerance, ed.), vol. 293 of *Lecture Notes in Computer Science*, pp. 120–127, Springer, 1988.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, vol. 32, pp. 372–382, Apr. 1985.

[11] O. Goldreich, *Foundations of Cryptography*, vol. I & II. Cambridge University Press, 2001–2004.

[12] S. Goldwasser, S. Micali, and R. L. Rivest, "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks," *SIAM Journal on Computing*, vol. 17, no. 2, pp. 281–308, 1988.

[13] V. Hadzilacos and S. Toueg, "Fault-Tolerant Broadcasts and Related Problems," *Distributed Systems (2nd Ed.)*, pp. 97–145, 1993.

[14] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The SecureRing Protocols for Securing Group Communication," in *Proc. 31st Annual Hawaii International Conference on System Sciences (HICSS)*, pp. 317–326, Jan. 1998.

[15] K. Kursawe and V. Shoup, “Optimistic Asynchronous Atomic Broadcast,” in *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP)* (L. Caires, G. F. Italiano, L. Monteiro, *et al.*, eds.), vol. 3580 of *Lecture Notes in Computer Science*, pp. 204–215, Springer, 2005.

[16] L. Lamport, “The Part-Time Parliament,” *ACM Transactions on Computer Systems*, vol. 16, pp. 133–169, May 1998.

[17] D. Malkhi, M. Merritt, and O. Rodeh, “Secure Reliable Multicast Protocols in a WAN,” *Distributed Computing*, vol. 13, pp. 19–28, Jan. 2000.

[18] M. O. Rabin, “Randomized Byzantine Generals,” in *Proc. 24th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 403–409, 1983.

[19] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders, “Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems,” in *Proc. International Conference on Dependable Systems and Networks (DSN-2002)*, pp. 229–238, June 2002.

[20] M. Reiter, “Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart,” in *Proc. 2nd ACM Conference on Computer and Communications Security*, pp. 68–80, 1994.

[21] M. K. Reiter, “The Rampart Toolkit for Building High-Integrity Services,” in *Theory and Practice in Distributed Systems*, vol. 938 of *Lecture Notes in Computer Science*, pp. 99–110, Springer, 1995.

[22] F. B. Schneider, “Implementing Fault-Tolerant Services using the State Machine Approach: A Tutorial,” *ACM Computing Surveys*, vol. 22, pp. 299–319, Dec. 1990.

[23] F. B. Schneider and L. Zhou, “Distributed Trust: Supporting Fault-Tolerance and Attack-Tolerance,” Tech. Rep. TR 2004-1924, Cornell Computer Science Department, Jan. 2004.

[24] V. Shoup, “Practical Threshold Signatures,” in *Advances in Cryptology: EUROCRYPT 2000* (B. Preneel, ed.), vol. 1087 of *Lecture Notes in Computer Science*, pp. 207–220, Springer, 2000.

[25] S. A. Vanstone, P. C. van Oorschot, and A. Menezes, *Handbook of Applied Cryptography*. CRC Press, 1996.