Parsimonious Asynchronous Byzantine-Fault-Tolerant Atomic Broadcast

HariGovind V. Ramasamy[1] and Christian Cachin[2]

[1] University of Illinois, Coordinated Science Laboratory, Urbana, IL 61801, USA. ramasamy@crhc.uiuc.edu.
[2] IBM Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland. cca@zurich.ibm.com.

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 [20] 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, 21]. 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.

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 [2, 7, 15] 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 model, in particular by assuming some degree of synchrony (like Rampart [18], SecureRing [13], and ITUA [16]). 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 a practical asynchronous atomic broadcast [5] protocol that has optimal resilience $t < n/3$, relying on a trusted initialization process and on public-key cryptography. The protocol involves one randomized Byzantine agreement [6] per round of atomically delivered payload messages.

The BFT protocol by Castro and Liskov [8] and the protocol by Kursawe and Shoup [14] (the *KS protocol* for short) 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. 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 protocols switch 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 $\mathcal{O}(n^2)$ 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. A comparison of our protocol with the asynchronous atomic broadcast protocols mentioned above is given in Table 1.

Our Approach. The starting point for the development of our protocol, which we call PABC, is the BFT protocol [8]. 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 [14] prevents this attack by ensuring that at least one payload is delivered during the recovery phase. It employs a round of randomized multi-valued Byzantine agreement (MVBA) 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 a consistent broadcast protocol, also known as *echo broadcast* [18]. The replacement directly leads to an amortized message complexity of only $\mathcal{O}(n)$. But consistent broadcast is weaker than reliable broadcast and guarantees agreement only among those correct parties that actually deliver the payload. Therefore, 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 delivered a payload from consistent broadcast, but no way for other correct parties to learn about this fact. We solve this problem by delaying the atomic delivery of a payload delivered from consistent broadcast until more payloads have been delivered from consistent broadcast. 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 atomically deliver the payloads up to that point; to implement this step, every party must

Protocol	Sync. for Safety?	Sync. for Liveness?	Public-key Operations?	Message Complexity	
				Normal	Worst-case
Rampart [19]	yes	yes	yes	$\mathcal{O}(n)$	unbounded
SecureRing [13]	yes	yes	yes	$\mathcal{O}(n)$	unbounded
ITUA [16]	yes	yes	yes	$\mathcal{O}(n)$	unbounded
Cachin et al. [5]	no	no	yes	exp. $\mathcal{O}(n^2)$	exp. $\mathcal{O}(n^2)$
BFT [8]	no	yes	no	$\mathcal{O}(n^2)$	unbounded
KS [14]	no	no	yes	$\mathcal{O}(n^2)$	exp. $\mathcal{O}(n^2)$
Protocol PABC	no	no	yes	$\mathcal{O}(n)$	exp. $\mathcal{O}(n^2)$

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

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 atomically deliver at least one payload, which guarantees that the protocol makes progress in every epoch.

Organization of the Paper. The rest of the paper is organized as follows. Section 2 introduces preliminaries, some protocol primitives on which our algorithm relies, and the definition of atomic broadcast. The protocol is presented in Section 3, and its practical significance is discussed in Section 4. For lack of space, the details of the formal system model and the analysis of our protocol are contained in the full version [17].

2 Preliminaries

2.1 System Model

This section contains an overview of the 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 are controlled by the adversary and are called *corrupted*; the other parties are called *correct*. We use a *static* corruption model, and there is an initialization algorithm run by a trusted *dealer* for system setup. All computations by the parties, the adversary, and the dealer are probabilistic, polynomial-time algorithms. Since our model is based on the formal approach in cryptography, we allow for a negligible probability of failure in the specification of our protocols. The system model includes a digital signature scheme that is secure against existential forgery using adaptive chosen-message attacks [11].

Each pair of parties is linked by an *authenticated asynchronous channel* that provides message integrity. Messages on the channels are scheduled by the adversary. However, we assume that every message on a channel between two correct parties is *eventually* delivered. Every protocol instance is identified by a unique string ID, called the *tag*. Formally, the local interface to our protocols consists of *input actions*, which are messages of the form $(ID, \texttt{in}, type, \ldots)$ and *output actions*, which are messages of the form $(ID, \texttt{out}, type, \ldots)$. The parties receive and generate *protocol messages* of the form $(ID, type, \ldots)$, which are delivered to other parties over the channels. Before a party starts to process messages for an instance ID, the instance with that ID must be *initialized*.

2.2 Protocol Primitives

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* [18]; 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*, the party 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.*

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 to allow 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 [22].

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 MVBA 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 $v \in \{0,1\}^*$ that satisfies 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*.*

We use the MVBA protocol of Cachin et al. [5], which has expected message complexity $\mathcal{O}(n^2)$ and expected communication complexity $\mathcal{O}\big(n^3 + n^2(K + L)\big)$, 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 [12], 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 [14]. We do not formalize their *fairness* condition, although Protocol PABC satisfies an equivalent notion.

3 The Parsimonious Atomic Broadcast Protocol

We now describe Protocol PABC in detail. The line numbers refer to the detailed protocol description in Figures 1–3.

3.1 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.

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: $complained \leftarrow$ `false` {indicates if this party already complained about P_l}
9: $start\_recovery \leftarrow$ `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: $m \leftarrow$ `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: $recovery()$

Figure 1: Protocol PABC for party P_i and tag ID (Part I)

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.

function $deliver(m)$**:**
39: **if** $m \neq$ `dummy` **then**
40: $remove(m, \mathcal{I})$
41: $\mathcal{D} \leftarrow \mathcal{D} \cup \{m\}$
42: output $(ID, \texttt{out}, \texttt{a-deliver}, m)$
upon receiving message $(ID, \texttt{complain}, e)$ from P_j for the first time:
43: $c \leftarrow c + 1$
44: **if** $(c = t + 1)$ **and** $\neg complained$ **then**
45: $complain()$
46: **else if** $c = 2t + 1$ **then**
47: $start\_recovery \leftarrow$ `true`
function $complain()$**:**
48: send $(ID, \texttt{complain}, e)$ to all parties
49: $complained \leftarrow$ `true`

predicate $Q_{ID|\texttt{watermark}.e}\Big(\big[(s_1, C_1, \sigma_1), \ldots, (s_n, C_n, \sigma_n)\big]\Big) \equiv$
(for at least $n - t$ distinct j, $s_j \neq \perp$) **and**
(for all $j = 1, \ldots, n$, it holds $(s_j = \perp)$ **or**
(σ_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 $ID|\texttt{bind}.e.s_j)$))

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

Figure 2: Protocol PABC for party P_i and tag ID (Part II)

The leader binds a sequence number to every payload that it receives in an `initiate` message, and conveys the binding 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 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).

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 48); it also sets the $complained$ flag (line 49) 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 Recovery Phase

The recovery phase consists of three parts: determining a watermark sequence number, synchronizing all parties up to the watermark, and 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.

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. 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

function *recovery*()**:**

{*Part 1: agree on watermark*}

50: compute a signature σ 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: $(s_j, C_j, \sigma_j) \leftarrow (\perp, \perp, \perp) \qquad (1 \leq j \leq n)$
53: **wait for** $n-t$ messages $(ID, \texttt{committed}, e, s_j, C_j, \sigma_j)$ from distinct P_j s.t. C_j completes the *sc-broadcast* instance $ID|\texttt{bind}.e.s_j$ and σ_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** $s' \leq \min\{s-1, w\}$ **do**
60: **if** $s' \geq 0$ **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** $s \leq w$ **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 σ on $(ID, \texttt{queue}, e, i, \mathcal{I})$
74: send the message $(ID, \texttt{queue}, e, i, \mathcal{I}, \sigma)$ to all parties
75: $(\mathcal{I}_j, \sigma_j) \leftarrow (\perp, \perp) \qquad (1 \leq j \leq n)$
76: **wait for** $n-t$ messages $(ID, \texttt{queue}, e, j, \mathcal{I}_j, \sigma_j)$ from distinct P_j s.t. σ_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 $ID|\texttt{deliver}.e$ 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 party P_i and tag ID (Part III)

$n-t$ 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 Analysis

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* $n > 3t$.

The proof can be found in the full version [17].

To analyze the complexity of Protocol PABC, 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 exchanges the *log* of the epoch and involves Byzantine agreement. Parts 1 and 3 use MVBA with proposal values of length $\mathcal{O}(n|m|)$; hence, the expected message complexity is $\mathcal{O}(n^2)$ and the expected communication complexity is $\mathcal{O}(n^3|m|)$. In part 2, $\mathcal{O}(n^2)$ `complete` messages are exchanged; 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 in the epoch; this leads to a communication complexity of $\mathcal{O}\big(n^2 B(|m| + K)\big)$.

Hence, 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.

3.4 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 3.1 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.

4 Practical Significance

In our formal system model, the adversary controls the scheduling of messages and hence the timeouts; 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. In contrast to the 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. Practical observations indicate that unstable network conditions are the exception rather than the norm. During periods of stability and when no new intrusions are detected, our protocol will make fast progress, but both safety and liveness are still guaranteed even if the network is unstable.

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 Intl. Workshop on Distributed Algorithms*, vol. 725 of *Lecture Notes in Computer Science*, pp. 189–203, 1993.
2. P. Berman and J. A. Garay, "Randomized Distributed Agreement Revisited," in *Proc. 23th Intl. Symp. Fault-Tolerant Computing*, pp. 412–419, 1993.
3. G. Bracha, "An Asynchronous $[(n-1)/3]$-Resilient Consensus Protocol," in *Proc. 3rd Symp. Principles of Distributed Computing*, pp. 154–162, 1984.
4. C. Cachin, "Distributing Trust on the Internet," in *Proc. Intl. Conf. Dependable Systems and Networks*, 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*, vol. 2139 of *Lecture Notes in Computer Science*, pp. 524–541, 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 Symp. Theory of Computing*, pp. 42–51, 1993.
8. M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Transactions on Computer Systems*, vol. 20, pp. 398–461, Nov. 2002.
9. Y. Desmedt, "Society and Group Oriented Cryptography: A New Concept," in *Advances in Cryptology: CRYPTO '87*, vol. 293 of *Lecture Notes in Computer Science*, pp. 120–127, 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. 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.
12. V. Hadzilacos and S. Toueg, "Fault-Tolerant Broadcasts and Related Problems," *Distributed Systems (2nd Ed.)*, pp. 97–145, 1993.
13. K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The SecureRing Protocols for Securing Group Communication," in *Proc. 31st Hawaii Intl. Conf. on System Sciences*, pp. 317–326, Jan. 1998.

14. K. Kursawe and V. Shoup, "Optimistic Asynchronous Atomic Broadcast," in *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, vol. 3580 of *Lecture Notes in Computer Science*, pp. 204–215, 2005.
15. M. O. Rabin, "Randomized Byzantine Generals," in *Proc. 24th Symp. Foundations of Computer Science*, pp. 403–409, 1983.
16. 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. Intl. Conf. Dependable Systems and Networks*, pp. 229–238, June 2002.
17. H. V. Ramasamy and C. Cachin, "Parsimonious asynchronous Byzantine-fault-tolerant atomic broadcast." Manuscript, available from the authors., Aug. 2005.
18. 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.
19. 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, 1995.
20. F. B. Schneider, "Implementing Fault-Tolerant Services using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, vol. 22, pp. 299–319, Dec. 1990.
21. F. B. Schneider and L. Zhou, "Distributed Trust: Supporting Fault-Tolerance and Attack-Tolerance," Tech. Rep. TR 2004-1924, Cornell University, Jan. 2004.
22. V. Shoup, "Practical Threshold Signatures," in *Advances in Cryptology: EUROCRYPT 2000*, vol. 1087 of *Lecture Notes in Computer Science*, pp. 207–220, 2000.