Optimistic Asynchronous Atomic Broadcast

Klaus Kursawe[1] and Victor Shoup[2]

[1] KU Leuven
[2] New York University

Abstract. This paper presents a new protocol for atomic broadcast in an *asynchronous* network with a maximal number of *Byzantine* failures. It guarantees both *safety* and *liveness* without making any timing assumptions. Under normal circumstances, the protocol runs in an extremely efficient "optimistic mode," while in rare circumstances the protocol may briefly switch to a less efficient "pessimistic mode."

1 Introduction

Atomic broadcast is a fundamental building block in fault tolerant distributed computing. By ordering broadcast requests in such a way that they are delivered in the same order to all honest recipients, a synchronization mechanism is provided that deals with many of the most problematic aspects of asynchronous networks. We present a new protocol for atomic broadcast in an *asynchronous* network with a maximal number of *Byzantine* failures. It guarantees both *safety* and *liveness* without making any timing assumptions or using any type of "failure detector," and under normal circumstances is just as efficient as a simple "Bracha broadcast."

The FLP "impossibility" result [F+85] implies that there is no deterministic protocol for Byzantine agreement (and hence, for atomic broadcast) that guarantees both safety and liveness. However, there are *randomized* protocols that terminate quickly with very high probability.

A protocol for asynchronous Byzantine agreement may be used as a building block for atomic broadcast. Canetti and Rabin's protocol [CR93] runs in polynomial time, but is in fact, highly impractical. The protocol of Cachin *et al.* [C+00] is a practical, polynomial-time protocol that makes use of public-key cryptographic primitives that can be proven correct in the "random oracle" model [BR93], assuming a computationally bounded adversary; this protocol relies on a trusted dealer during system set-up, but after this, an arbitrary number of instances of the protocol can be executed. Building on [C+00], the paper [C+01] presents a fairly practical protocol for atomic broadcast. However, this protocol still uses a lot of fairly expensive, public-key operations, and may not be fast enough for some applications.

Our protocol is inspired by the innovative work of Castro and Liskov [CL99b, CL99a, C00]. Like their protocol, our protocol works in two phases: an *optimistic phase* and a *pessimistic phase*. The optimistic phase is very "lightweight" —

L. Caires et al. (Eds.): ICALP 2005, LNCS 3580, pp. 204–215, 2005.

each request is processed using nothing more than a "Bracha broadcast" [B84] — in particular, no public-key cryptography is used (only message authentication codes, which are very cheap, are used). As long as the network is reasonably behaved, the protocol remains in the optimistic phase — even if some number of parties, barring a designated leader, are corrupted. If there are unexpected network delays, or the leader is corrupted, several parties may "time out," shifting the protocol into the pessimistic phase. The pessimistic phase is somewhat more expensive than the optimistic phase — both in terms of communication and computational complexity. Nevertheless, it is still reasonably practical, although certainly not as efficient as the optimistic phase. The pessimistic phase cleans up any potential "mess" left by the current leader, re-synchronizing the protocol, after which the optimistic phase starts again with a new leader.

The optimistic phase of our protocol is essentially the same as that of Castro and Liskov. While [CL99b] relies extensively on expensive public-key cryptographic operations, the optimized versions in [CL99a, C00] do not use on public-key cryptography in the optimistic phase. Therefore, we expect that in practice, our protocol is just as efficient as theirs. However, our pessimistic phase is quite different. In the Castro/Liskov protocol, the new leader is responsible for re-synchronizing the protocol; this re-synchronization may fail, either because the new leader is corrupt or because of unexpected network delays, in which case yet another leader must take on the task of re-synchronization. In contrast, in our protocol, this re-synchronization is done using a distributed computation, based on randomized Byzantine agreement, and is guaranteed to succeed, regardless of the behavior of the corrupted parties and regardless of any network delays.

Castro and Liskov's protocol is completely deterministic, and hence is subject to the FLP impossibility result. Indeed, although their protocol guarantees safety, it does not guarantee liveness, unless one makes additional *timing assumptions*. Our protocol guarantees both safety and liveness without making any timing assumptions *at all*, while being just as efficient in practice as the Castro/Liskov protocol. The trade-off, of course, is that our protocol is randomized and relies on more cryptographic assumptions; however, there is no practical downside to this, in terms of either security or performance, and so it seems to be a trade-off worth making.

Our work builds on the work of [C+01] in two ways: we use the same definitional framework as [C+01], and we make novel use of a protocol in [C+01] for *multivalued Byzantine agreement*.

Other Related Work. There is a rich literature on ordering broadcast channels, including several implementations and a broad theoretical basis. However, most work in the literature is done in the crash-failure model; much less work has been done in the Byzantine failure model. Rampart [R94] and SecureRing [K+98] directly transfer crash-failure protocols into the Byzantine setting by using a modified failure detector along with digital signatures. The disadvantage of this approach is that it is relatively expensive, as a large number of public-key cryptographic operations need to be performed. Furthermore, there are attacks on the failure detector [A+95] that can violate the safety of these protocols. Doudou

et al. [D+00] take a similar approach to that of Castro and Liskov. However, their protocol is described in a more abstract and modular way, leading to a protocol that is somewhat less complex and easier to analyze.

2 System Model and Problem Statement

Our formal system model and definitions of security are the same as in [C+00, C+01], which models attacks by *computationally bounded* adversaries. We refer the reader to [C+01] for complete details. We give only a brief summary here. We assume a network of n parties $P_1, \ldots, P_n$, t of which are corrupted and fully controlled by an adversary. We shall assume that $t < n/3$. We also assume a trusted dealer that is needed only at system set-up time. Informally, the adversary also has full control over the network; the adversary may insert, duplicate, and reorder messages at will.

More formally, at the beginning of the attack, the trusted dealer is run, initializing the internal state of the honest parties; the initial state information for the corrupted parties is given to the adversary. The attack then proceeds in steps. In each step of the attack, the adversary delivers a single message to an honest party, upon receipt of which the party updates its internal state and generates one or more response messages. These response messages indicate their origin and intended destination; however, the adversary is free to do with these messages what he wishes: to deliver them when he wishes, in any order that he wishes; he may also deliver them more than once, or not all. We do assume, however, that the adversary may not modify messages or "fake" their origin. This assumption is reasonable, since this property can be effectively enforced quite cheaply using message authentication codes.

We assume that the adversary's corruptions are *static*: the set of corrupted parties is chosen once and for all at the very beginning of the attack. However, it should be straightforward to prove that our protocol is secure in an *adaptive* corruption model, assuming all underlying cryptographic primitives are secure in this model.

Because we want to use cryptographic techniques, it does not make sense to consider "infinite runs" of protocols, but rather, we only consider attacks that terminate after some bounded amount of steps. The number of steps in the adversary's attack, as well as the computational complexity of the adversary, are assumed to be bounded by a polynomial in some security parameter.

Our protocols are defined such that they are only guaranteed to make progress to the extent to which the adversary actually delivers messages. To ensure that such a protocol behaves well in practice, an implementation would have to resend messages until receiving (secure) acknowledgments for them. We do not discuss any of these implementation details any further in this paper.

In our formal model, there is no notion of time. However, in making the transition from the optimistic phase to the pessimistic phase of our protocol, we need a way to test if an unexpectedly large amount of time has passed since some progress has been made by the protocol. That is, we need a "time out"

mechanism. This is a bit difficult to represent in a formal model in which there is no notion of time. Nevertheless, we can effectively implement such a "time out" as follows: to start a timer, a party simply sends a message to itself, and when this message is delivered to that party, the clock "times out." By representing time outs in this way, we effectively give the adversary complete control of our "clock."

We define the *message complexity* of a protocol as the number of messages generated by all honest parties. This is a random variable that depends on the adversary and the value of the security parameter, and is denoted $MC(ID)$, where ID identifies a particular protocol instance. The term *probabilistically uniformly bounded* is a technical term that we borrow from [C+01]. Let X be a random variable associated with a run of a protocol that depends on the adversary and the value of the security parameter. Informally, "X is probabilistically uniformly bounded" means that X is distributed "very tightly" around a quantity Y, where Y is bounded by a polynomial in the security parameter that is independent of the adversary. See [C+01] for the formal definition.

Our definition of atomic broadcast comes directly from [C+01], with just some minor notational changes. As we define it, an atomic broadcast primitive offers one or several broadcast channels, each specified by some channel identifier ID. Before a party can use a channel, it must be explicitly *opened*. Formally speaking, this is done by the adversary. At any point, the adversary may deliver the message $(ID, \texttt{in}, \texttt{a-broadcast}, m)$ to some honest party, where m is an arbitrary bit string (of bounded size); we say the party *a-broadcasts the request* m at this point. At any point, an honest party may generate an output message $(ID, \texttt{out}, \texttt{a-broadcast}, m)$, which is given to the adversary; we say the party *a-delivers the request* m at this point. We adopt the following terminological convention: a "request" is something that is *a-broadcast* or *a-delivered*, while a "message" is something that is sent or delivered in the implementation of the protocol.

To give higher level protocols the option to block the atomic broadcast protocol, the delivering party waits for an acknowledgment after every *a-delivery* of a request. That is, the number of *a-delivered* requests is equal to either the number of acknowledgments or the number of acknowledgments plus one. This is necessary so that higher-level protocols may satisfy a property analogous to the *efficiency* property (see Definition 1 below). Without this ability to synchronize protocol layers, a low-level atomic broadcast protocol could generate an arbitrary amount of network traffic without a higher-level protocol ever doing anything useful.

At any point in time, for any honest party P_i, we define $\mathcal{B}^{(i)}$ to be the set of requests that P_i has *a-broadcast*, and we define $\mathcal{D}^{(i)}$ to be the set of requests that P_i has *a-delivered*. We say that one request in $\mathcal{B}^{(i)}$ is *older* than another if P_i *a-broadcast* the first request before it *a-broadcast* the second request. At any point in time, we also define $\mathcal{D}^* = \cup_{\text{honest } P_i} \mathcal{D}^{(i)}$.

In discussing the values of the sets $\mathcal{B}^{(i)}$, $\mathcal{D}^{(i)}$, or $\mathcal{D}^*$ at particular points in time, we consider the sequence of events $E_1, \ldots, E_k$ during the adversary's

attack, where each event but the last is either an *a-broadcast* or *a-delivery* by an honest party, and the last event is a special "end of attack" event. The phrase "at time τ," for $1 \leq \tau \leq k$, refers to the point in time just *before* event E_τ occurs.

Definition 1 (Atomic Broadcast). *A protocol for* atomic broadcast *satisfies the following conditions, for all channels ID and all adversaries, with all but negligible probability.*

Agreement: *If some honest party has* a-delivered m *on channel ID, then all honest parties* a-deliver m *on channel ID, provided the adversary opens channel ID for all honest parties, delivers all associated messages, and generates acknowledgments for every party that has not yet* a-delivered m *on channel ID.*

Total Order: *Suppose one honest party has* a-delivered $m_1, \ldots, m_s$ *on channel ID, and another honest party has* a-delivered $m'_1, \ldots, m'_{s'}$ *on channel ID with* $s \leq s'$. *Then* $m_l = m'_l$ *for* $1 \leq l \leq s$.

Validity: *There are at most* t *honest parties* P_j *with* $\mathcal{B}^{(j)} \backslash \mathcal{D}^* \neq \emptyset$, *provided the adversary opens channel ID for all honest parties, delivers all associated messages, and generates all acknowledgments.*

Fairness: *There exist a quantity* Δ, *which is bounded by a fixed polynomial in the security parameter (independent of the adversary), such that the following holds. Suppose that at some time* τ_1, *there is a set* $\mathcal{S}$ *of* $t+1$ *honest parties, such that for all* $P_j \in \mathcal{S}$, *the set* $\mathcal{B}^{(j)} \backslash \mathcal{D}^*$ *is non-empty. Suppose that there is a later point in time* τ_2 *such that the size of* $\mathcal{D}^*$ *increases by more than* Δ *between times* τ_1 *and* τ_2. *Then there is some* $P_j \in \mathcal{S}$, *such that the oldest request in* $\mathcal{B}^{(j)} \backslash \mathcal{D}^*$ *at time* τ_1 *is in* $\mathcal{D}^*$ *at* τ_2.

Efficiency: *At any point in time, the quantity* $MC(ID)/(|\mathcal{D}^*|+1)$ *is probabilistically uniformly bounded.*

Integrity: *Every honest party* a-delivers *a request* m *at most once on channel ID. Moreover, if all parties follow the protocol, then* m *was previously* a-broadcast *by some party on channel ID.*

3 Protocol Conventions and Notations

At each step of an attack, the adversary delivers a message to an honest party, and activates the honest party: the party performs some computations, updates its internal state, generates messages, and then returns control to the adversary. Messages delivered to a party are appended to the rear of an *incoming message queue*. When activated, the party may examine this queue, and remove any messages it wishes.

A party consists of one or more threads of execution. When a party is activated, each thread is in a *wait state*, waiting for one of the corresponding conditions to be satisfied. If the condition upon which any thread is waiting is satsified, the corresponding thread is activated (if several threads could be activated, one is chosen arbitrarily), and this thread runs until it reaches another

wait state. This process continues until all threads are in wait states whose conditions are not satisfied, and then control returns to the adversary.

Our protocol syntax is rather self explaining, with one exception. A *wait* condition can either *receive* messages or *detect* them. In the former case, the messages are deleted from the queue (and thus do not trigger any further conditions) while in the latter, they remain in the queue. We also define an abstract timeout mechanism, which allows a process to start or stop the timer or wait for a timeout. In our model, this is implemented by the party simply sending a message to itself. As the adversary has full control over message delivery, this gives him full control over the timer, too.

4 Our New Protocol for Atomic Broadcast

The protocol operates in epochs, each epoch $e = 0, 1, 2$, etc., consisting of an optimistic and a pessimistic phase. In the optimistic phase, a designated leader orders incoming requests by assigning sequence numbers to them and initiating a Bracha broadcast [B84]; the optimistic phase guarantees the *agreement* and *total order* properties, but not the *validity* or *fairness* properties; however, the protocol can effectively determine if *validity* or *fairness* are potentially threatened, and if so, switch to the pessimistic phase, which cleans up any "mess" left by the current leader; then the optimistic phase starts again with a new leader.

4.1 Overview and Optimistic Phase

In the optimistic phase of epoch e, when a party *a-broadcasts* a request m, it *initiates* the request by sending a message of the form $(ID, \texttt{initiate}, e, m)$ to the leader for epoch e. When the leader receives such a message, it *0-binds* a sequence number s to m by sending a message of the form $(ID, \texttt{0-bind}, e, m, s)$ to all parties. Sequence numbers start at zero in each epoch. Upon receiving a *0-binding* of s to m, an honest party *1-binds* s to m by sending a message of the form $(ID, \texttt{1-bind}, e, m, s)$ to all parties. Upon receiving $n - t$ such *1-bindings* of s to m, an honest party *2-binds* s to m by sending a message of the form $(ID, \texttt{2-bind}, e, m, s)$ to all parties. A party also *2-binds* s to m if it receives $t + 1$ *2-bindings* of s to m — this has the effect of "amplifying" *2-bindings*, which is used to ensure *agreement*. Upon receiving $n - t$ such *2-bindings* of s to m, an honest party *a-delivers* m, provided all messages with lower sequence numbers were already delivered, enough acknowledgments have been received, and m was not already *a-delivered*.

A party only sends or reacts to *0-*, *1-*, or *2-bindings* for sequence numbers s in a "sliding window" $\{w, \ldots, w + \mathit{WinSize} - 1\}$, where w is the number of requests already *a-delivered* in this epoch, and *WinSize* is a fixed system parameter. Keeping the "action" bounded in this way is necessary to ensure *efficiency* and *fairness*.

The number of requests that any party *initiates* but has not yet *a-delivered* is bounded by a parameter *BufSize*: a party will not *initiate* any more requests

once this bound is reached. We denote by $\mathcal{I}$ the set of requests that have been *initiated* but not *a-delivered*, and we call this the *initiation queue*. If sufficient time passes without anything leaving the initiation queue, the party "times out" and *complains* to all other parties. These *complaints* are "amplified" analogously to the *2-bindings*. Upon receiving $n-t$ *complaints*, a party enters the pessimistic phase of the protocol. This strategy will ensure *validity*. Keeping the size of $\mathcal{I}$ bounded is necessary to ensure *efficiency* and *fairness*.

Also to ensure *fairness*, a party keeps track of the "age" of the requests in its initiation queue, and if it appears that the oldest request is being ignored, i.e., many other requests are being *a-delivered*, but not this one, then the party simply refuses to generate *1-bindings* until the problem clears up. If $t+1$ parties block in this way, they effectively prevent the remaining parties from making any progress in the optimistic phase, and thus, the pessimistic phase will be entered, where the fairness problem will ultimately be resolved.

We say that an honest party P_i *commits* s to m in epoch e, if m is the sth request (counting from 0) that it *a-delivered* in this epoch, optimistically or pessimistically.

Now the details. The state variables for party P_i are as follows.

Epoch number e**:** The current epoch number, initially zero.

Delivered set $\mathcal{D}$**:** All requests that have been *a-delivered* by P_i. It is required to ensure that requests are not *a-delivered* more than once; in practice, however, other mechanisms may be employed for this purpose. Initially, $\mathcal{D}$ is empty.

Initiation queue $\mathcal{I}$**:** The queue of requests that P_i *initiated* but not yet *a-delivered*. Its size is bounded by *BufSize*. Initially, $\mathcal{I}$ is empty.

Window pointer w**:** w is the number of requests that have been *a-delivered* in this epoch. Initially, $w = 0$. The optimistic phase of the protocol only reacts to messages pertaining to requests whose sequence number lies in the "sliding window" $\{w, \ldots, w + \mathit{WinSize} - 1\}$. Here, *WinSize* is a fixed system parameter.

Echo index sets BIND_1 **and** BIND_2**:** The sets of sequence numbers which P_i has *1-bound* or *2-bound*, respectively. Initially empty.

Acknowledgment count *acnt***:** Counts the number of acknowledgments received for *a-delivered* requests. Initially zero.

Complaint flag *complained***:** Set if P_i has issued a complaint. Initially *false*.

Initiation time $it(m)$**:** For each $m \in \mathcal{I}$, $it(m)$ is equal to the value of w at the point in time when m was added to $\mathcal{I}$. Reset to zero across epoch boundaries. These variables are used in combination with a fixed parameter *Thresh* to ensure *fairness*.

Leader index l**:** The index of the leader in the current epoch; we simply set $l = (e \bmod n) + 1$. Initially, $l = 1$.

Scheduled request set $\mathcal{SR}$**:** Only maintained by the current leader. It contains the set of messages which have been assigned sequence numbers in this epoch. Initially, it is empty.

Next available sequence number *scnt***:** Only maintained by the leader. Value of the next available sequence number. Initially, it is zero.

/* Initiate m**. */**
upon receiving a message $(ID, \texttt{in}, \texttt{a-broadcast}, m)$ for some m such that $m \notin \mathcal{I} \cup \mathcal{D}$ and $|\mathcal{I}| < \mathit{BufSize}$ (note that we take the *oldest* such message first):
Send the message $(ID, \texttt{initiate}, e, m)$ to the leader.
Add m to $\mathcal{I}$; set $it(m) \leftarrow w$.
/* 0-bind *scnt* **to** m**. */**
upon receiving a message $(ID, \texttt{initiate}, e, m)$ for some m, such that $i = l$ and $w \leq \mathit{scnt} < w + \mathit{WinSize}$ and $m \notin \mathcal{D} \cup \mathcal{SR}$:
Send the message $(ID, \texttt{0-bind}, e, m, \mathit{scnt})$ to all parties.
Increment *scnt* and add m to $\mathcal{SR}$.
/* 1-bind s **to** m**. */**
upon receiving a message $(ID, \texttt{0-bind}, e, m, s)$ from the current leader for some m, s such that $w \leq s < w + \mathit{WinSize}$ and $s \notin \mathit{BIND}_1$ and $((\mathcal{I} = \emptyset)$ or $(w \leq \min\{it(m) : m \in \mathcal{I}\} + \mathit{Thresh}))$:
Send the message $(ID, \texttt{1-bind}, e, m, s)$ to all parties; add s to BIND_1.
/* 2-bind s **to** m**. */**
upon receiving $n-t$ messages of the form $(ID, \texttt{1-bind}, e, m, s)$ from distinct parties that agree on s and m, such that $w \leq s < w + \mathit{WinSize}$ and $s \notin \mathit{BIND}_2$:
Send the message $(ID, \texttt{2-bind}, e, m, s)$ to all parties; add s to BIND_2.
/* Amplify a 2-binding of s **to** m**. */**
upon detecting $t+1$ messages of the form $(ID, \texttt{2-bind}, e, m, s)$ from distinct parties that agree on s and m, such that $w \leq s < w + \mathit{WinSize}$ and $s \notin \mathit{BIND}_2$:
Send the message $(ID, \texttt{2-bind}, e, m, s)$ to all parties; add s to BIND_2.
/* Commit s **to** m**. */**
upon receiving $n-t$ messages of the form $(ID, \texttt{2-bind}, e, m, s)$ from distinct parties that agree on s and m, such that $s = w$ and $\mathit{acnt} \geq |\mathcal{D}|$ and $m \notin \mathcal{D}$ and $s \in \mathit{BIND}_2$:
Output $(ID, \texttt{out}, \texttt{a-deliver}, m)$; increment w; add m to $\mathcal{D}$, and remove it from $\mathcal{I}$ (if present); **stop timer**.
/* Start timer. */
upon (timer not running) and (not *complained*) and $(\mathcal{I} \neq \emptyset)$ and $(\mathit{acnt} \geq |\mathcal{D}|)$:
start timer.
/* Complain. */
upon timeout:
if not *complained*: send the message $(ID, \texttt{complain}, e)$ to all parties; set *complained* $\leftarrow$ *true*.
/* Amplify complaint. */
upon detecting $t+1$ messages $(ID, \texttt{complain}, e)$ from distinct parties, such that not *complained*:
Send the message $(ID, \texttt{complain}, e)$ to all parties; set *complained* $\leftarrow$ *true*; **stop timer**.
/* Go pessimistic. */
upon receiving $n-t$ messages $(ID, \texttt{complain}, e)$ from distinct parties, such that *complained*:
Execute the procedure *Recover* below.

Fig. 1. The optimistic phase

The protocol for party P_i consists of two threads. The first is a trivial thread that simply counts acknowledgments for *a-delivered* requests; it consists of an infinite loop whose body is as follows:

wait until receiving an acknowledgment; increment *acnt*

The main thread is an infinite loop whose body is as follows:

case *MainSwitch* **end case**

where the *MainSwitch* is a sequence of **upon** clauses described in Figure 1.

4.2 Fully Asynchronous Recovery

The recovery protocol is invoked if the optimistic phase appears to not work properly; this happens if either the leader is faulty or the network is too slow. Its job is to synchronize the parties by *a-delivering* all broadcasts that any honest party *may* have already *a-delivered*, and to guarantee the efficiency of the overall protocol by assuring that *some* messages are *a-delivered*. Finally, it hands over to a new leader to restart a new optimistic phase.

Validated Multivalued Byzantine Agreement. Our recovery-protocol builds on top of validated multivalued Byzantine agreement (i.e., the agreement is not restricted to a binary value), as defined and implemented in [C+01]. The final agreement value must be legal according to some validation function, which guarantees that it is some "useful" value. The definition of the validation function is clear from the context as the exact form of a valid proposal is defined in the protocol description. In the atomic broadcast protocol, we use the phrase *"propose X_i for multivalued Byzantine agreement on X"* to denote the invocation of a validated multivalued Byzantine agreement protocol, where X_i is P_i's initial proposal, and X the resulting agreement value.

Overview of the Recovery Procedure. We distinguish between three types of requests: (i) requests for which it can be guaranteed that they have been *a-delivered* by an honest party; (ii) requests that potentially got *a-delivered* by an honest party; (iii) requests for which it can be guaranteed that they have not been *a-delivered* by an honest party. For the first two kinds of requests, an order of delivery might already be defined, and has to be preserved. The other requests have not been *a-delivered* at all, so the recovery protocol has complete freedom on how to order them. They can not be left to the next leader, however, as an adversary can always force this leader to be thrown out as well. To guarantee efficiency, the recovery procedure has to ensure that *some* request is *a-delivered* in every epoch. This is precisely the property that Castro and Liskov's protocol fails to achieve: in their protocol, without imposing additional timing assumptions, the adversary can cause the honest parties to generate an arbitrary amount of messages before a single request is *a-delivered*. According to the three types of requests, the recovery protocol consists of three parts.

Part 1: Requests whose sequence number is determined. A "watermark" $\hat{s}_e$ is jointly computed, which has the property that at least one honest party opti-

mistically committed the sequence number $\hat{s}_e$, and no honest party optimistically committed a sequence number higher than $\hat{s}_e + 2 \cdot \mathit{WinSize}$. After computing the watermark, all parties "catch up" to the watermark, i.e., commit all sequence numbers up to $\hat{s}_e$, by simply waiting for $t+1$ consistent *2-bindings* for each sequence number up to the watermark. The work performed in this part constant, and especially independent of the number of unfinished requests.

Part 2: Requests whose sequence number may be determined. Here, we deal with the requests that might or might not have been *a-delivered* by some honest party in the optimistic phase of this epoch. We have to ensure that if some honest party has optimistically *a-delivered* a request, then all honest parties *a-deliver* this request as well. The sequence numbers of requests with this property lie in the interval $\hat{s}_e + 1 \ldots \hat{s}_e + 2 \cdot \mathit{WinSize}$. Each party makes a proposal that indicates what action should be taken for all sequence numbers in this critical interval. Again, multivalued Byzantine agreement will be used to determine which of possibly several valid proposals should be accepted. While this part is relatively expensive, we can guarantee an upper bound of the number of requests processed here, which is determined by the window-size parameter.

Part 3: Undetermined Requests. This part is the one that guarantees that some messages are *a-delivered* in this epoch. We use a multivalued Byzantine agreement protocol to agree on a certain set of additional requests that should be *a-delivered* this epoch. We need to do this to ensure fairness and efficiency.

Terminology of the Recovery Procedure. For any party P_i, and any message α, we denote by $\{\alpha\}_i$ a signed version of the message, i.e., α concatenated with a valid signature under P_i's public key on α, along with P_i's identity.

For any $s \geq -1$, a *strong consistent set Σ for s* is a set of $t+1$ correctly signed messages from distinct parties, each of the form $\{(ID, \texttt{s-2-bind}, e, s')\}_j$ for some j and $s' \geq s$.

A *valid watermark proposal* $\mathcal{M}$ is a set of $n-t$ correctly signed messaged from distinct parties, each of the form $\{(ID, \texttt{watermark}, e, \Sigma_j, s_j)\}_j$ for some j, where Σ_j is a strong consistent set of signatures for s_j. The maximum value s_j appearing in these watermark messages is called the *maximum sequence number* of $\mathcal{M}$.

For any $s \geq 0$, a *weak consistent set Σ' for s* is a set of $n-t$ correctly signed messages from distinct parties — each of the form $\{(ID, \texttt{w-2-bind}, e, s, m_j)\}_j$ for some j — such that either all $m_j = \perp$ (indicating no *2-binding* for s), or there exists a request m and all m_j are either m or $\perp$. In the former case, we say Σ' *defines* $\perp$, and in the latter case, we say Σ' *defines* m.

A *valid recover proposal* $\mathcal{P}$ is a set of $n-t$ correctly signed messages from distinct parties each of the form $\{(ID, \texttt{recover-request}, e, \mathcal{Q}_j)\}_j$ for some j, where $\mathcal{Q}_j$ is a set of at most *BufSize* requests.

The protocol for the pessimistic phase is presented in Figure 2.

A proof of security of the complete atomic broadcast protocol, as well as a number of other details, can be found in the full version of this paper [KS01].

/* Part 1: Recover requests with a determined sequence-number */

Send a the signed message $\{(ID, \texttt{s-2-bind}, e, \max(BIND_2 \cup \{-1\}))\}_i$ to all parties.

wait until receiving a strong consistent set Σ_i for $w-1$.

Send the signed message $\{(ID, \texttt{watermark}, e, \Sigma_i, w-1)\}_i$ to all parties.

wait until receiving a valid watermark proposal $\mathcal{M}_i$.

Propose $\mathcal{M}_i$ for multivalued Byzantine agreement on a valid watermark proposal $\mathcal{M}$.

Set $\hat{s}_e \leftarrow \tilde{s} - \mathit{WinSize}$, where $\tilde{s}$ is the maximum sequence number of $\mathcal{M}$.

while $w \leq \hat{s}_e$ do:

- **wait until receiving** $t+1$ messages of the form $(ID, \texttt{2-bind}, e, m, w)$ from distinct parties that agree on m, such that $\mathit{acnt} \geq |\mathcal{D}|$.
- Output $(ID, \texttt{out}, \texttt{a-deliver}, m)$; increment w.
- Add m to $\mathcal{D}$, and remove it from $\mathcal{I}$ (if present).

/* Part 2: Recover requests with a potentially determined sequence-number */

For $s \leftarrow \hat{s}_e + 1$ to $\hat{s}_e + (2 \cdot \mathit{WinSize})$ do:

- If P_i sent the message $(ID, \texttt{2-bind}, e, m)$ for some m, set $\tilde{m} \leftarrow m$; otherwise, set $\tilde{m} \leftarrow \perp$.
- Send the signed message $(ID, \texttt{w-2-bind}, e, s, \tilde{m})$ to all parties.
- **wait until receiving** a weak consistent set Σ'_i for s.
- Propose Σ'_i for multivalued Byzantine agreement on a weak consistent set Σ' for s.
- Let Σ' define m.
- If ($s \geq w$ and $m \in \mathcal{D}$) or $m = \perp$, exit the for loop and go to Part 3.
- If $m \notin \mathcal{D}$: **wait until** $\mathit{acnt} \geq |\mathcal{D}|$; output $(ID, \texttt{out}, \texttt{a-deliver}, m)$; increment w; add m to $\mathcal{D}$, and remove it from $\mathcal{I}$ (if present).

/* Part 3: Recover undetermined Requests */

If $w = 0$ and $\mathcal{I} \neq \emptyset$: send the message $(ID, \texttt{recover-help}, e, \mathcal{I})$ to all parties.

If $w = 0$ and $\mathcal{I} = \emptyset$: **wait until receiving** a message $(ID, \texttt{recover-help}, e, \mathcal{Q})$, such that $\mathcal{Q}$ is a non-empty set of at most $\mathit{BufSize}$ requests, and $\mathcal{Q} \cap \mathcal{D} = \emptyset$.

If $w \neq 0$ or $\mathcal{I} \neq \emptyset$: set $\mathcal{Q} \leftarrow \mathcal{I}$.

Send the signed message $\{(ID, \texttt{recover-request}, e, \mathcal{Q})\}_i$ to all parties.

wait until receiving a valid recover proposal $\mathcal{P}_i$.

Propose $\mathcal{P}_i$ for multivalued Byzantine agreement on a valid recover proposal $\mathcal{P}$.

Sequence through the request set of $\mathcal{P}$ in some deterministic order, and for each such request $m \notin \mathcal{D}$, do the following:

- **wait until** $\mathit{acnt} \geq |\mathcal{D}|$; output $(ID, \texttt{out}, \texttt{a-deliver}, m)$; increment w; add m to $\mathcal{D}$, and remove it from $\mathcal{I}$ (if present).

/* Start New Epoch */

Set $e \leftarrow e+1$; $l \leftarrow (e \bmod n) + 1$; $w \leftarrow \mathit{scnt} \leftarrow 0$; $\mathcal{SR} \leftarrow BIND_1 \leftarrow BIND_2 \leftarrow \emptyset$; $\mathit{complained} \leftarrow \mathit{false}$.

For each $m \in \mathcal{I}$: send the message $(ID, \texttt{initiate}, e, m)$ to the leader; set $it(m) \leftarrow 0$.

Fig. 2. The pessimistic phase

References

[A+95] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Tech. Rep. TR95-1534, Cornell University, Computer Science Department, 1995.

[BR93] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *1st ACM Conf. on Computer and Communications Security*, pp. 62–73, 1993.

[B84] G. Bracha. An asynchronous $[(n-1)/3]$-resilient consensus protocol. In *Proc. of the 3rd Ann. ACM Symp. on Principles of Distributed Computing*, pp. 154–162, 1984.

[C00] M. Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, Massachusetts Institute of Technology, 2000.

[C+01] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology—Crypto 2001*, pp. 524–541, 2001.

[C+00] C. Cachin, K. Kursawe, and V. Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement using Cryptography. In *ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pp. 123–132, 2000.

[CL99a] M. Castro and B. Liskov. Authenticated byzantine fault tolerance without public-key cryptography. Tech. Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science, 1999.

[CL99b] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd Symp. Operating Systems Design and Implementation*, 1999.

[CR93] R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proc. 25th Ann. ACM Symp. on Theory of Computing*, pp. 42–51, 1993.

[D+00] Doudou, Guerraoui, and Garbinato. Abstractions for devising byzantine-resilient state machine replication. In *SRDS: 19th Symp. on Reliable Distributed Systems*, 2000.

[F+85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[K+98] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *31st Hawaii International Conference on System Sciences*, pp. 317–326, 1998.

[KS01] K. Kursawe, V. Shoup. Optimistic asynchronous atomic broadcast. Cryptology ePrint Archive, Report 2001/022, `http://eprint.iacr.org`, 2001.

[R94] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proc. of the 2nd ACM Conference on Computer and Communication Security*, pp. 68–80, 1994.