RZ 3344 (# 93390) 05/21/01
Computer Science 20 pages

Research Report

Optimistic Asynchronous Atomic Broadcast

Klaus Kursawe and Victor Shoup

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

Optimistic Asynchronous Atomic Broadcast

Klaus Kursawe and Victor Shoup

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

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 or using any type of "failure detector," and its *amortized* message and computational complexity is essentially the same as that of a simple "Bracha broadcast." Under normal circumstances, the protocol runs in an "optimistic mode," with extremely low message and computational complexity — essentially, just performing a Bracha broadcast for each request. In particular, no "expensive" public-key cryptographic operations are used. In rare circumstances, the protocol may briefly switch to a "pessimistic mode," where both the message and computational complexity are significantly higher than in the "optimistic mode," but are still reasonable.

Keywords: Asynchronous Consensus, Byzantine Faults, Atomic Broadcast, State Machine Replication

1 Introduction

Atomic Broadcast is a fundamental building block in fault tolerant distributed computing. By ordering broadcast requests in such a way that two broadcast requests are received in the same order by 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 its *amortized* message and computational complexity is essentially the same as that of a simple "Bracha broadcast."

The FLP "impossibility" result [FLP85] implies that there is no protocol for Byzantine agreement that runs in an *a priori* bounded number of steps, and guarantees both safety and liveness. Moreover, it is fairly well known that Byzantine agreement and atomic broadcast are equivalent, so that any protocol for solving atomic broadcast could be used to solve Byzantine agreement, and *vice versa*. However, this impossibility result does not rule out *randomized* protocols where the *expected* number of steps is bounded.

There are a few protocols for asynchronous Byzantine agreement. Canetti and Rabin [CR93] present a polynomial-time protocol for asynchronous Byzantine agreement; however, their protocol cannot be used in practice, because of its enormous message complexity. Cachin *et al.* [CKS00] give a fairly practical protocol for asynchronous Byzantine agreement that makes use of public-key cryptographic primitives that can be proven secure in the "random oracle" model [BR93]. Building on [CKS00], the paper [CKPS01] presents a fairly practical protocol for ataomic broadcast. While in some settings, the protocol in [CKPS01] may be adequate, in other settings, it may not. In particular, because of its heavy reliance on public-key cryptography, the protocol in [CKPS01] can easily become "compute bound."

Our protocol is inspired by the innovative work of Castro and Liskov [CL99b, CL99a, Cas00]. Like their protocol, our protocol works in two phases: an *optimistic phase* and a *pessimistic phase*. The optimistic phase is very "lightweight" — each request is processed using nothing more than a "Bracha broadcast" ([Bra84]) — in particular, no public-key cryptography is 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, 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. Therefore, we expect that in practice, our protocol is just as efficient as theirs. However, our pessimistic phase is quite different, and makes use of randomized Byzantine agreement as well as some additional public-key cryptographic operations. The pessimistic phase of Castro and Liskov makes use of public-key cryptography as well, and it is not clear if their pessimistic phase is significantly more or less efficient than ours — determining this would require some experimentation.

Castro and Liskov's pessimistic 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 is randomized, and it guarantees both safety and liveness without making any timing assumptions at all, and without relying on any kind of "failure detector." This is a not just a theoretical issue: if the timing mechanism does not work properly in Castro and Liskov's protocol, the protocol may cycle indefinitely, without doing anything useful, whereas in our protocol, the performance "gracefully" degrades.

1.1 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 [Rei94] and SecureRing [KMMS98] directly transfer crash-failure protocols into the Byzantine world 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 expensive cryptographic operations need to be computed. Furthermore, there are attacks on the failure detector [ACBMT95] that can violate the safety of these protocols.

The BFS system by Castro and Liskov [CL99b] addresses these problems. As already mentioned, they only require timing assumptions to guarantee liveness, while the safety properties of the protocol hold regardless of timing issues. A similar approach is taken by Doudou *et al.* [DGG00], but their protocol is described and analyzed in terms of a Byzantine failure detector.

While both [CL99b] and [DGG00] still rely extensively on expensive public-key cryptographic operations, the extension of BFS in [CL99a, Cas00] relies much less on public-key cryptography.

2 System Model and Problem Statement

2.1 Formal System Model

Our formal system model and definitions of security are taken from [CKS00, CKPS01], which models attacks by *computationally bounded* adversaries. We refer the reader to [CKPS01] for complete details. We give only a brief summary here.

We assume a network of n parties $P_1, \ldots, P_n$, up to $t < \frac{n}{3}$ of which are corrupted and fully controlled by an adversary. Informally, the adversary also has full control over the network; the adversary may insert, duplicate, and reorder messages at will.

More formally, an attack 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 if corrupted parties is chosen once and for all at the very beginning of the attack. Making this assumption greatly simplifies the security analysis, and allows one to make use of certain cryptographic primitives that could not otherwise be proven secure.

Although we have not done so, we believe it should be straightforward to prove that our atomic broadcast protocol is secure in a *dynamic* corruption model, assuming all underlying cryptographic primitives are secure in this model (in particular, the common coin as used in [CKS00, CKPS01]).

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. In 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 represent 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."

2.2 Some Technicalities

As already mentioned above, there is a security parameter $\lambda = 0, 1, 2 \ldots$ that is used to instantiate a protocol instance. All adversaries and protocols can be modeled as Turing machines that run in time bounded by a polynomial in λ. We make the convention that the parameter n is bounded by a fixed polynomial in λ, independent of the adversary. We make a similar assumption on the sizes of all messages in the protocol: excessively large messages are simple never generated by or delivered to honest parties.

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 λ. We denote it by $MC(ID)$, where ID identifies a particular protocol instance.

We say that a function ϵ, mapping non-negative integers to non-negative reals, is *negligible* if for all c there exists $k_0(c)$ such that for all $k \geq k_0(c)$, $\epsilon(k) \leq k^{-c}$.

We say that some quantity is *negligible,* if it is bounded by a negligible function in λ.

For a given protocol, a *protocol statistic* X is a family of real-valued, non-negative random variables $\{X_A(\lambda)\}$, parameterized by adversary A and security parameter λ, where each $X_A(\lambda)$ is a random variable on the probability spaced induce by A's attack on the protocol with security parameter λ. We call X a *bounded protocol statistic* if for all adversaries A, there exists a polynomial p_A such that for all $\lambda \geq 0$, $0 \leq X_A(\lambda) \leq p_A(\lambda)$. The message complexity $MC(ID)$ is an example of a bounded protocol statistic.

A bounded protocol statistic X is called *uniformly bounded (by p)* if there exists a fixed polynomial p such that for all adversaries A, there is a negligible function ϵ_A such that for all $\lambda \geq 0$, $\Pr[X_A(\lambda) > p(\lambda)] \leq \epsilon_A(\lambda)$.

A bounded protocol statistic X is called *probabilistically uniformly bounded (by p)* if there exists a fixed polynomial p and a fixed negligible function δ, such that for all adversaries A, there is a negligible function ϵ_A such that for all $k, \lambda \geq 0$, $\Pr[X_A(\lambda) \geq kp(\lambda)] \leq \delta(k) + \epsilon_A(\lambda)]$.

If X is probabilistically uniformly bounded by p, then for all adversaries A, $E[X_A(\lambda)] = O(p(\lambda))$, where the big-'O' constant is independent of the adversary. Additionally, if Y is probabilistically uniformly bounded by q, then $X \cdot Y$ is probabilistically uniformly bounded by $p \cdot q$, and $X + Y$ is probabilistically uniformly bounded by $p + q$. Thus, probabilistically uniformly bounded statistics are closed under polynomial composition, which makes them useful for analyzing the composition of several protocols. The same observations apply to uniformly bounded statistics as well.

2.3 Formal Definition of Atomic Broadcast

Our definition of atomic broadcast comes directly from [CKPS01], with some modification.

As we define it, an atomic broadcast primitive offers one or several broadcast channels, specified by some channel identifier *ID*. It guarantees that if some – not necessarily honest – parties broadcast requests, they arrive at all (honest) recipients in the same order. Furthermore, the channels are reliable, i.e., all honest parties receive the same set of messages.

At some 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 some point, an honest party may generate an output message $(ID, \texttt{out}, \texttt{a-broadcast}, m)$. We say the party *a-delivers the request* m at this point.

As a matter of terminology, we adopt the following convention: a "request" is something that is *a-broadcast* or *a-delivered*, while a "message" is something that is 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 delivery of a request. That is, the number of *a-delivered* requests exceeds the number of acknowledgments by at most one. This is necessary to allow higher level protocols to be efficient (according to the definition below); without this ability to synchronize between 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*. At any point in time, we also define $\mathcal{D}^* = \cup_{\text{honest } P_i} \mathcal{D}^{(i)}$.

For an honest party P_i, 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.

Recall that $MC(ID)$ is the message complexity of a protocol.

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 an honest party P_i has *a-delivered* $m_1, \ldots, m_s$ on channel ID, a distinct honest party P_j has *a-delivered* $m'_1, \ldots, m'_{s'}$ on channel ID, and $s \leq s'$. Then $m_l = m'_l$ for $1 \leq l \leq s$.

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

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

Validity: There are at most t honest parties P_j with $\mathcal{B}^{(j)} \backslash \mathcal{D}^{(j)} \neq \emptyset$, 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 (i.e., just before the τ_1-st time that an honest party *a-broadcasts* or *a-delivers* a request), there is a set $\mathcal{S}$ of $t + 1$ 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 time 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.

These security requirements are the same as those in [CKPS01], except in two ways. First, our validity condition is somewhat relaxed. Our notion of validity requires $t+1$ honest parties to *a-broadcast* a request m, to ensure that m is eventually *a-delivered*, whereas the stronger notion of validity in [CKPS01] requires only one party to do so.

We have chosen to achieve only this weaker notion of validity for several reasons. First, for many applications, this condition is sufficient: a client who sends a request to all parties can be sure that the request is eventually *a-delivered* by all honest parties. Second, our protocol is simpler than it would have to be in order to ensure the stronger notion of validity. Third, it is possible to transform any protocol that satisfies weak validity to one that satisfies the stronger notion of validity, by layering a fairly simple protocol on top of the original protocol.

Second, our notion of fairness is somewhat stronger than in [CKPS01]. Our definition of fairness allows one to analyze composed protocols more easily than those in [CKPS01]. Validity and fairness complement one another: validity ensures that a request that is *a-broadcast* by $t + 1$ parties is *a-delivered* provided all messages and acknowledgments are delivered, and fairness implies that such a request is *a-delivered* reasonably quickly if is *a-delivered* at all.

Note that for the rest of the paper, the term *efficiency* will be used to refer to above technical condition.

3 Validated Multivalued Byzantine Agreement

Our 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 [CKPS01]. Similarly to atomic broadcast, every instance of such a protocol has a particular ID. As opposed to some protocols in the literature [Rab83, TC84], the agreement protocol we need is not allowed to fall back on a default value; the final agreement value must be legal according to some verification, which guarantees that it is some "useful" value. To ensure this, validated multivalued Byzantine agreement has a global, polynomial-time computable predicate Q_{ID} known to all parties, which is determined by a higher-level application (in this case, the atomic broadcast protocol). Each party may propose a value v together with a proof π that should satisfy Q_{ID}.

Definition 2 (Validated Multivalued Byzantine Agreement). A protocol solves *validated Byzantine agreement* it satisfies the following conditions for all adversaries, except with negligible probability:

External Validity: Any honest party that terminates for ID decides v validated by π such that $Q_{ID}(v, \pi)$ holds.

Agreement: If some honest party decides v for ID, then any honest party that terminates decides v for ID.

Liveness: If all honest parties have been activated on ID and all associated messages have been delivered, then all honest parties have decided for ID.

Efficiency: $varMC(ID)$ is probabilistically uniformly bounded.

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. The definition of Q_{ID} and π is clear from the context.

4 Protocol conventions and notations

In this section, we give a brief description of a formal model of the internal structure of an honest party, and introduce some notation for describing the behavior of an honest party.

Recall that with each step of an attack, the adversary delivers a message to an honest party; the honest party then performs some computations, updating its internal state, and possibly generating messages. 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.

There may be several threads of execution for a given party, but at any point in time, at most one is active.

When a party is activated, all threads are in *wait states*. A wait state specifies a condition defined on the incoming message queue and other local state variables. If one or more threads are in a wait state whose condition is satisfied, one such thread is scheduled (arbitrarily, if more than one) to execute, and this thread runs until it reaches another wait state. This process continues until no threads are in a wait state whose condition is satisfied, and then the activation of the party is terminated, and control returns to the adversary. Of course, we restrict ourselves to polynomial-time protocols that always relinquish control to the adversary.

That completes the brief description of our formal model of the internal structure of an honest party. We now introduce the pseudo-code notation we will use to describe how a thread enters a wait state.

To enter a wait state, a thread may execute command **wait until** *condition*. Here, *condition* may be an ordinary predicate on state variables. Upon executing this statement, a thread enters a wait state with the given *condition*.

We also may specify a *condition* of the form **receiving** *messages*. In this case, *messages* describes a set of of one or more messages satisfying a certain predicate, possibly involving other state variables. Upon executing this statement, a thread enters a wait state, waiting for the arrival of messages satisfying the given predicate; moreover, when this predicate becomes satisfied, the matching messages are *moved* out of the incoming message queue, and into local state variables. Unless otherwise specified, if there is a single message that satisfies the predicate, then the first such (i.e., oldest) message in the incoming message queue is selected; otherwise, no particular order is guaranteed.

We also may specify a *condition* of the form **detecting** *messages*. The semantics of this are the same as for **receiving** *messages*, except that the matching messages are *copied* from the incoming message queue into local state variables.

In addition to waiting on a single condition, a thread may wait on a number of conditions simultaneously by executing the statement:

$$\textbf{case upon } \textit{condition}_1 : \textit{action}_1 \ ; \ \cdots \ \textbf{ upon } \textit{condition}_k : \textit{action}_k \ ; \textbf{end case}$$

Here, each $\textit{condition}_i$ as a *condition* as above, and each $\textit{action}_i$ is an action to be executed when the condition is satisfied. If more than one condition is satisfied, then an arbitrary choice is made.

We also define an abstract timeout mechanism. Each thread has a *timer*. One can view the *timer* as a special state variable that takes on two values: *stopped* and *running*. Initially, the timer is *stopped*. A thread may change this value by executing **start timer** or **stop timer** commands. A thread may also simply inspect the value of the *timer*. A thread may also execute a **wait until** or **case** statement with the *condition* **timeout**. Additionally, when the adversary activates a party, instead of delivering a message, it may deliver a *timeout signal* to a thread whose timer is *running*; when this happens, the timer is stopped, and if that thread is waiting on a *timeout*, the thread is activated.

This abstract timer can be implemented quite easily in our formal system model in §2.1 that does not include a timer mechanism. A **start timer** command is implemented by sending a unique message to oneself, and the adversary delivers a timeout signal by delivering this message.

Of course in a practical implementation, the timeout mechanism is implemented by using a real clock. However, by giving the adversary complete control over the timeout mechanism in our formal model, we effectively make no assumptions about the accuracy of the clock, except that the clock runs forward.

5 Our New Protocol for Atomic Broadcast

The protocol operates in epochs, each epoch $e = 1, 2$, etc., consisting of an optimistic and a pessimistic phase. In the optimistic phase, a designated leader is responsible to order incoming requests by assigning sequence numbers to them and initiating a reliable broadcast *a la* [Bra84]; the protocol only ensures that a dishonest leader cannot violate *total order*. If enough parties suspect that there is a threat to the validity or fairness of the protocol, they can shift the protocol into the pessimistic phase. The pessimistic phase cleans up any potential "mess" left by the current leader, after which the optimistic phase starts again with a new leader.

5.1 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 *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" $\{w, \dots, w + \mathit{WinSize} - 1\}$, where w is the number of requests already *a-delivered* in this epoch, and *WinSize* is a parameter. Keeping the "action" bounded 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 stop 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 $\mathcal{D}$**.** 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 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 $\mathit{it}(m)$**.** For each $m \in \mathcal{I}$, $\mathit{it}(m)$ 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 $l = (e \bmod n) + 1$.

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.

The protocol for party P_i consists of two threads. The first is a trivial thread that simply counts acknowledgments:

> **loop forever**
>
>   **wait until receiving** an acknowledgment: increment *acnt*.
>
> **end loop**

The main thread is as follows:

> **loop forever**
>
>   **case**
>
>   *MainSwitch*
>
>   **end case**
>
> **end loop**

where the *MainSwitch* is a sequence of **upon** clauses we now describe.

/* **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 $|\mathcal{I}| < \mathit{BufSize}$:

Send the message $(ID, \texttt{initiate}, e, m)$ to the leader.
Add m to $\mathcal{I}$.
Set $\mathit{it}(m) \leftarrow w$.

/* **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 $(w \leq \min\{\mathit{it}(m) : m \in \mathcal{I}\} + \mathit{Thresh}))$:

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* then:

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.

5.2 Fully Asynchronous Recovery

The recovery protocol tidies up all requests that where initiated under a (potentially) faulty leader. We distinguish between three types of requests:

- Requests for which it can be guaranteed that they have been *a-delivered* by an honest party.
- Requests that potentially got *a-delivered* by an honest party.
- 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. In the first part, a watermark $\hat{s}_e$ is jointly computed such that all requests with sequence numbers up to this $\hat{s}_e$ have been *a-delivered* by *some* honest party. The watermark has the property that at least one honest party *a-delivered* request $\hat{s}_e$, and no honest party *a-delivered* a request higher than $\hat{s}_e + 2 \cdot \mathit{WinSize}$.

The watermark is determined as follows. When P_i enters the pessimistic phase of the protocol, it sends out a signed statement to all parties that indicates its highest *2-bound* sequence number. Then, P_i waits for $t+1$ signatures on sequence numbers s' such that s' is greater than or equal to the highest sequence number s that P_i committed during the optimistic phase. Let us call such a set of signatures a *a strong consistent set of signatures for* s. Since P_i already received $n-t$ *2-bindings* for s, it is assured that at least $t+1$ of these came from honest parties, and so it will eventually receive a strong consistent set of signatures for s. Any party that is presented with such a set of signatures can conclude the following: one of these signatures is from an honest party, therefore some honest party sent a *2-binding* for a sequence number at least s, and therefore, because of the logic of the sliding window, that honest party committed $(s - \mathit{WinSize})$ in its optimistic phase.

Once P_i has obtained its own strong consistent set for s, it signs it and sends this signed strong consistent set to all parties, and collects a set $\mathcal{M}_i$ of $n-t$ signed strong consistent sets from other parties. Then P_i runs a multivalued Byzantine agreement protocol with input $\mathcal{M}_i$, obtaining a common set $\mathcal{M}$ of $n-t$ signed strong consistent sets. The watermark computed as $\hat{s}_e = (\tilde{s} - \mathit{WinSize})$, where $\tilde{s}$ is the maximum sequence number $\tilde{s}$ for which $\mathcal{M}$ contains a strong consistent set for $\tilde{s}$. We will show that no honest party commits a sequence number higher than $\hat{s}_e + (2 \cdot \mathit{WinSize})$ in its optimistic phase. And as already argued above, at least one honest party commits $\hat{s}_e$ in its optimistic phase.

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. By the logic of the protocol, since one of these *2-bindings* must come from an honest party, the correct request is *a-delivered*. Since one honest party has already committed s in its optimistic phase, at least $t+1$ honest parties have already sent corresponding *2-bindings*, and these will eventually arrive.

Part 2. In the second part, 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 *a-delivered* a request, then all honest parties do so. 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 is used to determine which of possibly several valid proposals should be accepted.

To construct such a proposal for sequence number s, each party P_i does the following. Party P_i sends out a signed statement indicating if it sent a *2-binding* for that s, and if so, the corresponding request m. Then P_i waits for a set of $n-t$ "consistent" signatures for s, such that the set does not contain conflicting requests. By the logic of the protocol, an honest party will eventually obtain such a consistent set, which we call a *weak consistent set of signatures for* s. If all signatures in this set are on statements that indicate no *2-binding*, then we say the set *defines no request*; otherwise, we say it *defines request* m, where m is the unique request appearing among the signed statements in the set. P_i's proposal consists of a set of weak consistent set of signatures for s. Any party that is presented with such a set can conclude the following: if the set defines no request, then no party optimistically commits s; if the set defines m, then *if* any honest party optimistically commits s to some m', then $m = m'$. Note that if the set defines some request m, this does not imply that s was committed optimistically, and indeed, if s was not optimistically committed, then the adversary can construct sets that define different requests.

Part 3. In the third part, we use a multivalued Byzantine agreement protocol to agree on a set of additional requests that should be *a-delivered* this epoch. This set will include the (possibly empty) initiation queues of at least $n-2t$ distinct honest parties. This property will be used to ensure fairness. Also, this set is guarantee to be non-empty if no requests were previously *a-delivered* (optimistically or otherwise) in this epoch. This property will be used to ensure efficiency.

5.2.1 The recovery procedure

We begin with some terminology.

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

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 $m_j = \bot$, or there exists a request m and all m_j are either m or $\bot$. In the former case, we say say Σ' *defines* $\bot$, and in the latter case, we say Σ' *defines* m.

For a set $\mathcal{Q}$ of requests and an integer $k \geq 0$, a $(\mathcal{Q}, k)$*-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 with $\mathcal{Q}_j \cap \mathcal{Q} = \emptyset$; moreover, if $k = 0$, we require that some $\mathcal{Q}_j$ is non-empty. We define the *request set* for $\mathcal{P}$ as the set of all requests that appear in any of the sets $\mathcal{Q}_j$.

/* Part 1: Recover Potentially delivered Requests */

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 $acnt \geq |\mathcal{D}|$.
    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 potentially delivered Requests */

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 $\tilde{m} \leftarrow \bot$.
    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 $m = \bot$, exit the for loop and go to Part 3.
    If $m \notin \mathcal{D}$ then:
        wait until $acnt \geq |\mathcal{D}|$.
        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 undelivered Requests */

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

wait until receiving a valid $(\mathcal{D}, w)$-recover proposal $\mathcal{P}_i$.
Propose $\mathcal{P}_i$ for multivalued Byzantine agreement on a valid $(\mathcal{D}, w)$-recover proposal $\mathcal{P}$.
Sequence through the request set of $\mathcal{P}$ in some deterministic order, and for each such request m, do the following:

wait until $acnt \geq |\mathcal{D}|$.
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 $e \leftarrow e + 1$.
Set $l \leftarrow (e \bmod n) + 1$.
Set $\mathcal{SR} \leftarrow BIND_1 \leftarrow BIND_2 \leftarrow \emptyset$.
Set $complained \leftarrow false$.
Set $w \leftarrow scnt \leftarrow 0$.
For each $m \in \mathcal{I}$:

Send the message $(ID, \texttt{initiate}, e, m)$ to the leader.
Set $it(m) \leftarrow 0$.

6 Analysis

If honest party P_i enters epoch e, let $\mathcal{D}_e^{(i)}$ denote the sequence of requests that honest party P_i *a-delivered* at the point in time where it entered this epoch. We say *consensus holds on entry to epoch* e if for any two honest parties P_i and P_j that enter epoch e, $\mathcal{D}_e^{(i)} = \mathcal{D}_e^{(j)}$. If consensus holds on entry to epoch e, and any honest party does enter epoch e, we denote by $\mathcal{D}_e$ the common value of the $\mathcal{D}_e^{(i)}$, and we denote by N_e the length of $\mathcal{D}_e$.

Recall that 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. If this occurs in the optimistic phase, we say P_i *optimistically commits* s to m.

Lemma 1. *In any epoch, if two honest parties* 2-bind *a sequence number s, then they* 2-bind *s to the same request.*

Moreover, if for some s, m, m', one honest party receives a set of $t + 1$ 2-bindings *of s to m and one honest party (possible the same one) receives a set of* $t + 1$ 2-bindings *of s to m', then $m = m'$.*

Proof. This is a fairly standard argument. If some honest party *2-binds* s to m, then some honest party (not necessarily the same one) has receives $n - t$ *1-bindings* of s to m. But since any two sets of $n - t$ parties must contain a common honest party, and no party *1-binds* a sequence number more than once, if one honest party receives $n - t$ *1-bindings* of s to m, and another receives $n - t$ *1-bindings* of s to m', then $m = m'$. That proves the first statement.

The second statement follows from the first, and the fact that any set of $t + 1$ parties must contain an honest party. □

Lemma 2. *If all all honest parties have entered epoch e, and all messages and timeouts have been delivered, and one honest party enters the pessimistic phase of the protocol in this epoch, then all honest parties have gone pessimistic in epoch e.*

Proof. An honest party enters the pessimistic phase of an epoch if it receives $n-t$ complaint messages. This implies that at least $t+1$ honest parties have sent a complaint message, thus every honest party will eventually receive at least $t+1$ complaint messages. This will cause all honest parties to send out complaint messages, thus all honest parties eventually receive at least $n-t$ complaints and thus will go pessimistic. □

Lemma 3. *Suppose that consensus holds on entry to some epoch e, that some honest party has entered this epoch, and that no honest party has gone pessimistic in this epoch. The the following conditions hold.*

local consistency: *If some honest party commits s to m, any honest party that also commits s, also commits s to m.*

local completeness: *If some honest party commits s to m, and all messages and timeouts have been delivered, and all honest parties have entered epoch e and received $N_e + s$ acknowledgments, then all honest parties have committed s.*

local validity: *If all messages, timeouts, and acknowledgments have been delivered, and all honest parties have entered epoch e, then at most t honest parties have non-empty initiation queues.*

local unique delivery: *Any honest party* a-delivers *each request at most once in this epoch.*

Proof. If some honest party commits s to m, then it has received $n-t$ *2-bindings* of s to m. At least $t+1$ of these are from honest parties. Moreover, by Lemma 1, any set of $t+1$ consistent *2-bindings* for s that an honest party receives are *2-bindings* to s.

Local consistency is now immediate.

If *local completeness* does not hold, let us choose s to be the minimal s for which this it does not hold.

Consider any honest party P_i. We want to show that in fact, P_i has committed s, yielding a contradiction.

By the minimality of s, it is easy to verify that the local value of w for any honest party P_j is at least s. Since $t+1$ honest parties have *2-bound* s to m, these *2-bindings* will be received at a point in time where s lies in P_j's window. So if P_j will itself *2-bind* s to m. Therefore all honest parties have *2-bound* s to m, and P_i has received these *2-bindings* while s was in its sliding window. Because consensus holds on entry to epoch e, and by the *consistency* part of this lemma, and by the minimality of s, it follows that all honest parties' $\mathcal{D}$ sets are equal at the point in time when $w = s$ (locally), and in particular $m \notin \mathcal{D}$ at this point in time, and so is not "filtered out" as a duplicate. Also, P_i has received sufficient acknowledgments, and so commits s to m.

Suppose *local validity* does not hold. Then the $t+1$ honest parties would certainly have sent complaint messages, and it is easy to verify that this would eventually cause all parties to complain, and hence go pessimistic. This contradicts our assumption that no party has gone pessimistic.

Unique delivery is clear from inspection, as duplicates are explicitly "filtered" in the optimistic phase. □

Lemma 4. *If all honest parties have entered the pessimistic phase of epoch e, and all messages and timeouts have been delivered, then all honest parties have agreed on a watermark $\hat{s}_e$.*

Proof. When an honest party P_i enters Part 1 of the pessimistic phase in some epoch, it will eventually obtain a strong consistent set Σ_i for $w-1$. To see this, observe that when P_i waits for strong consistent set Σ_i, it has already *a-delivered* sequence number $w-1$, and hence has received $n-t$ *2-bindings* for $w-1$. Of these, at least $t+1$ came from honest parties who, when they eventually enter the pessimistic phase for this epoch, will send an *s-2-bind* message with a sequence number at least $w-1$. These $t+1$ *s-2-bind* messages form a strong consistent set for $w-1$.

Thus, all honest parties eventually obtain strong consistent sets, and send corresponding watermark messages. Thus, all honest parties eventually obtain valid watermark proposals, and enter the multivalued Byzantine agreement with these proposals, and so by the liveness property of Byzantine agreement, all parties eventually agree on a common watermark proposal $\mathcal{M}$ with maximum sequence number $\tilde{s} = \hat{s}_e + \mathit{WinSize}$. □

Lemma 5. *If some honest party has computed $\hat{s}_e$, then*

(i) some honest party has optimistically committed $\hat{s}_e$, and

(ii) no honest party has optimistically committed a sequence number $\hat{s}_e + 2 \cdot \mathit{WinSize} + 1$.

Proof. Let $\tilde{s} = \hat{s}_e + \mathit{WinSize}$. To prove (i), note that $\mathcal{M}$ contains a strong consistent set for $\tilde{s}$. The existence of a strong consistent set for $\tilde{s}$ implies that at least one honest party *2-bound* $\tilde{s}$, which implies that this party has optimistically committed $\hat{s}_e$, because of the sliding window logic.

To prove (ii), suppose some honest party P_j optimistically commits $\hat{s}_e + 2 \cdot \mathit{WinSize} + 1 = \tilde{s} + \mathit{WinSize} + 1$. Then by the logic of the optimistic protocol, P_j must have received $n-t$ *2-bindings* for $\tilde{s} + \mathit{WinSize} + 1$, and so there must be a set $\mathcal{S}$ of $t+1$ honest parties who sent these *2-bindings*. By the logic of the sliding window, each party in $\mathcal{S}$ has optimistically committed $\tilde{s}+1$, and so has sent out a strong consistent set for a sequence number greater than $\tilde{s}$. By a standard counting argument, $\mathcal{M}$ must contain a contribution from some member of $\mathcal{S}$, and therefore the maximum sequence number of $\mathcal{M}$ is greater than $\tilde{s}$, which is a contradiction. □

Lemma 6. *Suppose $\hat{s}_e$ has been computed by some honest party. Let s be in the range $\hat{s}_e + 1 \ldots \hat{s}_e + 2 \cdot \mathit{WinSize}$.*

(i) If all honest parties generate w-2-bind *messages for s, these messages form weak consistent set for s.*

(ii) If one honest party optimistically commits s to m, then any weak consistent set for s defines m.

Proof. Part (i) follows directly from Lemma 1.

To prove (ii), if an honest party optimistically committed s to m in epoch e, then he received $t+1$ *2-bindings* of s to m from honest parties. Any set of $n-t$ *w-2-bind* messages must contain a contribution from one of these $t+1$ parties, and hence defines m. □

Lemma 7. *Suppose that consensus holds on entry to some epoch e, and that some honest party has entered the pessimistic phase in this epoch.*

local consistency: *If some honest party commits s to m, any honest party that also commits s, also commits s to m.*

local completeness: *If some honest party commits s to m, and all messages and timeouts have been delivered, and all honest parties have entered epoch e and received $N_e + s$ acknowledgments, then all honest parties have committed s.*

local validity: *If all messages, timeouts, and acknowledgments have been delivered, and all honest parties have entered epoch e, then all parties have entered epoch $e + 1$.*

boundary consistency: *If some honest party P_i commits s in epoch e, and some honest party P_j has entered epoch $e + 1$, then P_j commits s in epoch e.*

$e + 1$ **consensus:** *Consensus holds on entry to epoch $e + 1$.*

boundary completeness: *If some honest party enters epoch $e + 1$, and all messages and timeouts have been delivered, and all honest parties have entered epoch e and received $N_{e+1} - 1$ acknowledgments, then all honest parties have entered epoch $e + 1$.*

at least one delivery: *If some party enters epoch $e + 1$, then $N_{e+1} > N_e$ (i.e., at least one request is delivered in epoch e).*

local unique delivery: *Any honest party* a-delivers *each request at most once in this epoch.*

Proof (sketch). The same proof in the *local consistency* part of Lemma 3 implies in this case as well that any two parties that optimistically commit s, commit s to the same request.

If one honest party goes pessimistic, then by Lemma 2, all honest parties eventually go pessimistic. By Lemma 4, all honest parties eventually compute a common watermark $\hat{s}_e$.

By Lemma 5, part (i), all parties will eventually move through the loop in Part 1 of the pessimistic phase. To see this, note that since some honest party has optimistically *committed* s for all s up to $\hat{s}_e$, $t+1$ honest parties have *2-bound* s to m, and so when these *2-bindings* are delivered to any honest party, that party can *commit* s. Note also that these *commitments* are consistent, and no party *a-delivers* a request twice, since we are only delivering requests that have been optimistically *a-delivered*, and these are guaranteed to be consistent and duplicate-free.

By Lemma 6, part (i), all parties will eventually move through the loop in Part 2 of the pessimistic phase, since all of the weak consistent sets that they need will eventually be available. Lemma 5, part (ii), and Lemma 6, part (ii), together imply that any request that is optimistically *a-delivered* by some honest party will be *a-delivered* in Part 2 of the pessimistic phase in the same order by all honest parties.

Note that on entry to Part 3, consensus holds: all honest parties have exactly the same value $\mathcal{D}$ as they reach this point. If no requests were *a-delivered* either optimistically or in Parts 1 or 2, then all honest parties will expect a *recover proposal* in Part 3 containing a non-empty *recover request*. This will ensure that at least one request is *a-delivered* in this epoch, but one has to check that all honest parties will eventually receive a non-empty *recover request* in this case. To see why this is so, note that one honest party, say P_i, must have timed out while holding a non-empty initiation queue (otherwise, no party could have gone pessimistic). But since no requests were *a-delivered* prior to Part 3, P_i's *recover request* is non-empty. Thus, all honest parties move through Part 3 of the pessimistic phase consistently and without obstruction.

All of the claims in the lemma can be easily verified, given the above discussion. □

Lemma 8. *The* fairness *condition of Definition 1 holds with* $\Delta = \mathit{WinSize} + \mathit{Thresh} + 2 \cdot \mathit{PBound}$*, where* $\mathit{PBound} = 2 \cdot \mathit{WinSize} + (n-t) \cdot \mathit{BufSize}$*.*

Proof. Observe that *PBound* is an upper bound on the number of requests that can be *a-delivered* by any honest party in Parts 2 and 3 of the pessimistic phase of the protocol.

At any time τ, let us define $\mathcal{D}^*(\tau)$ to be the value of $\mathcal{D}^*$ at time τ. Also, define $e_{max}(\tau)$ to be the maximum value of e for any honest party at time τ.

Suppose that at some time τ_0, there is a set $\mathcal{S}$ of $t+1$ honest parties such that for all $P_j \in \mathcal{S}$, the sets $\mathcal{B}^{(j)} \backslash \mathcal{D}^*$ are non-empty at time τ_0. For each P_j in $\mathcal{S}$, let m_j denote the oldest request in $\mathcal{B}^{(j)} \backslash \mathcal{D}^*$ at time τ_0.

Clearly, either m_j lies in P_j's initiation queue at time τ_1, or P_j is currently in the pessimistic phase of some epoch, its initiation queue is empty, and m_j will enter its initiation queue as soon as P_j enters its next epoch.

Consider any point in time $\tau_1 > \tau_0$ such that $|\mathcal{D}^*(\tau_1) - \mathcal{D}^*(\tau_0)| = \mathit{PBound}$. If some m_j is in $\mathcal{D}^*(\tau_1)$, we are done; so we assume from now on that no m_j is in $\mathcal{D}^*(\tau_1)$.

If some honest party is in the pessimistic phase of epoch $e_{max}(\tau_0)$ at time τ_0, then since $|\mathcal{D}^*(\tau_1) - \mathcal{D}^*(\tau_0)| = \mathit{PBound}$, we must have $e_{max}(\tau_1) > e_{max}(\tau_0)$. Therefore, for all parties in $P_j \in \mathcal{S}$ such that P_j is in epoch $e_{max}(\tau_1)$ at time τ_1, it must hold that m_j is in P_j's initiation queue at time τ_1.

At any point in time after τ_1, if m_j lies in P_j's initiation queue, the value of $it(m_j)$ is the minimum among all requests in its initiation queue.

We define the quantity it_{max} as follows: if no party in $\mathcal{S}$ is in epoch $e_{max}(\tau_1)$ at time τ_1, then it_{max} is 0; otherwise, it_{max} is the maximum value of $it(m_j)$ for any party P_j in $\mathcal{S}$ that is in epoch $e_{max}(\tau_1)$ at time τ_1.

An honest party that *a-delivers* "too many" requests, none of which lie in its initiation queue, will refuse to send *1-bindings*. The precise statement of this is as follows.

Consider any point in time $\tau_2 > \tau_1$. For any party $P_j \in \mathcal{S}$, if P_j has not *a-delivered* m_j at time τ_2, then P_j has not generated any *1-bindings* in in epoch $e_{max}(\tau_1)$ for sequence numbers $it_{max} + \mathit{WinSize} + \mathit{Thresh}$ or above at time τ_2.

Further suppose that at time τ_2, no m_j is in $\mathcal{D}^*(\tau_2)$. Then we claim that no party has entered epoch $e_{max}(\tau) + 1$. To see this, note that in Part 3 of the pessimistic phase, since a valid *recover proposal* must contain contributions from $n-t$ parties, one of these must come from a party P_j in $\mathcal{S}$, who would have contributed a *recover request* containing m_j. Also, since no party P_j in $\mathcal{S}$ issued *1-bindings* for sequence numbers $it_{max} + \mathit{WinSize} + \mathit{Thresh}$ or above, no honest party could have optimistically committed such a sequence number. Therefore, $|\mathcal{D}^*(\tau_2) - \mathcal{D}^*(\tau_1)| \le \mathit{WinSize} + \mathit{Thresh} + \mathit{PBound}$.

That proves the lemma. □

We now state and prove our main theorem.

Theorem 9. *Our protocol satisfies the properties in our Definition 1 for atomic broadcast.*

Proof. We first define some auxiliary notions.

Let us say that an honest party P_i *globally commits a sequence number s to a request m*, if m is the sth request (counting from zero) *a-delivered* by P_i.

We then define *consistency*, *completeness*, and *unique delivery* as follows.

consistency: If some honest party globally commits s to m, any honest party that also globally commits s, also globally commits s to m.

completeness: If some honest party globally commits s to m, and all messages and timeouts have been delivered, and all honest parties have received s acknowledgments, then all honest parties have globally committed s.

unique delivery: Any honest party *a-delivers* each request at most.

It is clear that *consistency* and *completeness* hold if and only if *agreement* and *total order* (from Definition 1) hold.

One can prove by a completely routine induction argument, using Lemmas 7 and 3, that *consistency*, *completeness*, *validity*, *unique delivery* hold.

Integrity trivially follows from *unique delivery* and by simple inspection of the protocol.

Efficiency is also follows from the *at least one delivery* property in Lemma 7, and by simple inspection of the protocol.

Fairness follows from Lemma 8. □

References

[ACBMT95] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical Report TR95-1534, Cornell University, Computer Science Department, August 25, 1995.

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

[Bra84] G. Bracha. An asynchronous $[(n-1)/3]$-resilient consensus protocol. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 154–162, Vancouver, B.C., Canada, 27–29 August 1984.

[Cas00] M. Castro. *Practical Byzantine Fault Tolerance.* PhD thesis, Massachusetts Institute of Technology, November 2000.

[CKPS01] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. Research Report RZ 3317, IBM Research, 2001.

[CKS00] C. Cachin, K. Kursawe, and V. Shoup. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement using Cryptography. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 123–132, 16–19 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, June 1999.

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

[CR93] R. Canetti and T. Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC93*, pages 42 – 51, 1993.

[DGG00] Doudou, Guerraoui, and Garbinato. Abstractions for devising byzantine-resilient state machine replication. In *SRDS: 19th Symposium on Reliable Distributed Systems*. IEEE Computer Society Press, 2000.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[KMMS98] 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*, pages 317–326, Kona, Hawaii, January 1998. IEEE.

[Rab83] M. O. Rabin. Randomized Byzantine generals. In *24th Annual Symposium on Foundations of Computer Science*, pages 403–409, Tucson, Arizona, 7–9 November 1983. IEEE.

[Rei94] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, pages 68–80, November 1994.

[TC84] R. Turpin and B. A. Coan. Extending binary Byzantine Agreement to multivalued Byzantine Agreement. *Information Processing Letters*, 18(2):73–76, February 1984.