Università della Svizzera italiana

Faculty of Informatics

USI Technical Report Series in Informatics

Optimistic Atomic Multicast

Carlos Eduardo Bezerra[1,3], Fernando Pedone[1], Benoît Garbinato[2], Cláudio Geyer[3]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland
[2] Département des systèmes d'information, HEC, Université de Lausanne, Switzerland
[3] Instituto de Informática, Universidade Federal do Rio Grande do Sul, Brazil

Abstract

Message ordering is one of the cornerstones of reliable distributed systems. However, some ordering guarantees, such as atomic order, are expensive to implement in terms of message delays. This technical report presents Optimistic Atomic Multicast, a protocol that combines reduced latency and increased throughput. Messages can be delivered optimistically in a single communication step and conservatively in three communication steps. Differently from previous optimistic group communication protocols, Optimistic Atomic Multicast does not rely on spontaneous message ordering for fast delivery. In addition to presenting Optimistic Atomic Multicast, we provide detailed performance results comparing it to other ordering protocols in both local-area and wide-area networks.

Report Info

Published
February 2013

Number
USI-INF-TR-2013/01

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

Message ordering is one of the cornerstones of reliable distributed systems and lies at the core of fundamental fault-tolerant approaches such as state-machine replication [1, 2]. In state-machine replication, servers are replicated so that the failure of one or more replicas does not prevent client requests from being executed against non-faulty replicas. State-machine replication requires that (i) every correct replica receive all requests and (ii) no two replicas disagree on the order of received requests. These requirements are captured by group communication primitives that ensure total order, such as atomic broadcast [3].

Atomic broadcast simplifies the design of reliable distributed systems. However, it is typically expensive to implement (e.g., in terms of communication steps) and does not scale well (i.e., with the number of nodes involved in the ordering protocol). The first shortcoming is particularly problematic if the application is geographically distributed since it has been shown that atomic broadcast requires three communication steps [4]. The second shortcoming implies that throughput, measured in number of requests ordered by time unit, cannot be increased by adding nodes to the system, an inherent limitation of total order.

The distributed systems research community has been long aware of these problems and solutions to each one of them have been proposed. Atomic broadcast protocols with reduced latency have been designed based on optimistic assumptions about spontaneous message ordering (e.g., [5, 6, 7]), relaxed ordering requirements, dependent on application semantics (e.g., [8]), or both (e.g., [9, 10]). Atomic multicast addresses the (lack of) scalability of atomic broadcast. While atomic broadcast propagates messages to all system members, atomic multicast divides the members into groups and propagates messages to some groups only, not all. If the atomic multicast protocol is *genuine* [11], then only the message's addressees (i.e., a set of groups) and sender will coordinate to ensure order. As a result, atomic multicast can scale better than atomic broadcast. This is beneficial for applications whose state can be decomposed into sub-states, each one associated to a group (e.g., [12]) since throughput can be increased by multicasting requests to sets of groups, not the whole system.

This technical report presents Optimistic Atomic Multicast, the first atomic multicast protocol to exploit optimism, and thus, to combine reduced latency and increased throughput. Messages can be delivered optimistically in a single communication step and conservatively in three communication steps. To deliver messages optimistically, a process estimates a wait window based on expected message transmission delays and skews between node clocks. If the process' estimate holds, the order in which messages are delivered optimistically (i.e., the optimistic order) will match the conservative order.

Optimistic Atomic Multicast is *quasi-genuine*: not only the message's sender and destinations coordinate to establish order, but other groups may participate too. The set of groups that participate in the ordering of a message is determined by a relation, defined by the application. Moreover, this relation may change over time, a reconfiguration, so groups can be added and removed on-the-fly, reflecting the application needs. Reconfigurations are expected to be used sparingly though since they incur additional communication.

The contributions of this technical report are: (1) An atomic multicast protocol that delivers messages in three communication delays. (2) A variation of this protocol that improves latency and delivers messages optimistically in a single communication delay. (3) A detailed experimental assessment of these protocols both in local-area and wide-area networks.

The remainder of the technical report is organized as follows. Section 2 presents the system model and some definitions. Section 3 introduces Baseline Atomic Multicast. Section 4 contains the optimistic version of the algorithm. Section 5 describes reconfiguration. Section 6 presents our prototype and Section 7 evaluates its performance. Section 8 reviews related works. Section 9 concludes the report. A proof of correctness of our protocols is presented in the Appendix.

2 System model and definitions

In the following, we introduce the underlying system model (Section 2.1) and define the consensus abstraction (Section 2.2), atop of which we have designed our atomic multicast protocols.

2.1 Processes and communication

We consider a system $\Pi = \{p_1, ..., p_n\}$ of processes that communicate through message passing and do not have access to a shared memory or a global clock. The system is asynchronous: messages may experience arbitrarily large (although finite) delays and there is no bound on relative process speeds. We assume the benign crash-stop failure model: processes may fail by crashing, but do not behave maliciously. A process that does not crash is *correct*; otherwise it is *faulty*. We define $\Gamma = \{g_1, ..., g_m\}$ as the set of process groups in the system. Groups are disjoint, non-empty, and satisfy $\bigcup_{g \in \Gamma} g = \Pi$. For brevity, we write "$p \in \gamma$" instead of "$\exists g \in \gamma : p \in g$", where γ is any set of groups such that $\gamma \subseteq \Gamma$.

Processes can also communicate using fifo reliable multicast, defined by the primitives fr-mcast(γ, m) and fr-deliver(m), where m is a message and γ is the set of groups m is addressed to. Message m has two fields: $m.src$, m's source group, and $m.data$, the application-specific content. Fifo reliable multicast guarantees that (i) if a correct process p fr-mcasts m, then eventually all correct processes $q \in \gamma$ fr-deliver m *(validity)*; (ii) if a correct process p fr-delivers m, then eventually every correct process $q \in \gamma$ fr-delivers m *(agreement)*; (iii) for any process p and any message m, p fr-delivers m at most once, and only if $p \in \gamma$ and m was previously fr-mcast *(uniform integrity)*; and (iv) if p fr-mcasts m to γ and then m' to γ', then no $q \in \gamma \cap \gamma'$ fr-delivers m' without first fr-delivering m *(fifo order)*.

We say that a group g executes the actions of sending, receiving, multicasting and delivering a message with the meaning that one or more processes in g execute the action.

2.2 Consensus

An important part of this work relies on consensus to ensure that processes agree upon which messages are delivered and in which order they are delivered. Thus, we assume that consensus can be solved within a group. Moreover, we distinguish multiple instances of consensus executed within the same group with unique natural numbers. Consensus is defined by the primitives $propose_g(k, v)$ and $decide_g(k, v)$, where g is a group, k a natural number and v a value. Consensus satisfies the following properties in each instance k: (i) if process $p \in g$ decides v, then v was previously proposed by some process in g *(uniform integrity)*; (ii) if $p \in g$ decides v, then all correct processes in g eventually decide v *(uniform agreement)*; and (iii) every correct process in g eventually decides exactly one value *(termination)*.

3 Baseline Atomic Multicast

In this section we define the atomic multicast problem (Section 3.1), introduce the Baseline Atomic Multicast algorithm (Sections 3.2 and 3.3) and discuss its liveness guarantees (Section 3.4).

3.1 Problem definition

Atomic multicast allows messages to be sent to a set γ of groups. It is defined by the primitives multicast(γ, m) and deliver(m), where m is a message with fields $m.src$ and $m.data$, as defined in Section 2.1. Atomic multicast guarantees the following properties:

(i) If a correct process p multicasts m, then every correct process $q \in \gamma$ delivers m *(validity)*.

(ii) If p delivers m, then every correct process $q \in \gamma$ delivers m *(uniform agreement)*.

(iii) For any message m, every process $p \in \gamma$ delivers m at most once, and only if some process has multicast m previously *(uniform integrity)*.

(iv) If p multicasts m and then m' to groups γ and γ', respectively, then no process q in both γ and γ' delivers m' before delivering m *(fifo order)*.

(v) No two processes p and q in both γ and γ' deliver m and m' in different orders *(atomic order)*.

Atomic multicast generalizes atomic broadcast, where every message is multicast to all groups. To implement atomic multicast using an atomic broadcast algorithm, it suffices to broadcast every message m to all groups, and those not interested in m discard it. Obviously, this is not efficient. To rule out this kind of implementation, an early work introduced the notion of genuine atomic multicast algorithms [11], which spare unnecessary communication: to deliver message m multicast to γ, groups g and h only communicate if they are "concerned by m". Group x is concerned by m if the sender of m is in x or $x \in \gamma$. While genuineness is an important property for atomic multicast protocols, it is expensive: no genuine multicast protocol can deliver messages in fewer than two inter-group network delays [13]. Since we seek communication-efficient algorithms, we introduce next the concept of quasi-genuine atomic multicast protocols.

In a quasi-genuine atomic multicast protocol, the groups that communicate to deliver a message m are determined by an *a priori sendersTo* relation, defined based on the application semantics and access patterns—as opposed to m's sender and destination groups. For any group g, we define *sendersTo*(g) as the set of groups that can multicast a message to g. If m is multicast to a set of groups γ, then all groups in *sendersTo*(g), where $g \in \gamma$, can communicate to deliver m, even if some group h in *sendersTo*(g) is neither m's sender nor one of m's destination groups. We denote the *sendersTo* relation the system's *configuration*. A *sendersTo* relation can be changed via a reconfiguration, described in Section 5.

Applications whose data can be partitioned according to access locality can benefit from a quasi-genuine protocol. Consider a database divided in partitions P_1 and P_2 such that some data items are exclusively in P_1, some are exclusively in P_2, and some items are shared by both partitions—in other words, *the partitions are not disjoint*. Most commands access (i.e., read and write) items in one partition only and few commands include shared items. In this context, we create groups g_1, g_2 and g_3 such that processes in g_1 and g_2 replicate exclusive items in P_1 and P_2, respectively, and processes in g_3 replicate items shared by P_1 and P_2. A command C that accesses items in partition P_x, where $x \in \{1,2\}$, is handled by a process p in group g_x. If C accesses items exclusive to P_x only, then p multicasts C to g_x; if C includes exclusive and shared items in P_x, p multicasts C to g_x and g_3; finally, if C accesses only shared items, p multicasts C to g_3. Consequently, *sendersTo*(g_1) $= \{g_1\}$, *sendersTo*(g_2) $= \{g_2\}$ and *sendersTo*(g_3) $= \{g_1, g_2\}$. This scheme allows commands that access items stored exclusively in P_1 to be ordered independently of commands that access items stored exclusively in P_2.

3.2 Overview of the algorithm

Hereafter, we assume that each message m has two additional fields, $m.ts$, the message's timestamp, and $m.dst$, the set of groups m is addressed to. To multicast m to γ, process p in group g sets $m.ts$ to a unique *initial timestamp* and then fr-mcasts m to all processes in g. Such timestamp is created based on p's wallclock (i.e., p's real-time local clock) and p's unique identifier.

When processes in g fr-deliver m, they run a consensus instance within g to agree on m's *final timestamp*, after possibly adjusting it to ensure the following invariant: for any two messages m and m', multicast by processes in g, if a process in g decides m before m', then $m.ts < m'.ts$. This is important since we intend to deliver messages according to their final timestamp order. The initial timestamp assigned by the message's sender may violate the invariant due to the asynchrony of the system. If so, processes reassign the message's timestamp after consensus to keep the invariant.

Once group g has decided on m's final timestamp, m is fr-mcast to all its destination groups γ. Let h be a group in γ. To ensure that no message m' with a smaller final timestamp than m's (i.e., $m'.ts < m.ts$) is delivered after m, process $q \in h$ delivers m only when it has received at least one message from every group x in $sendersTo(h)$ with a timestamp greater than m's timestamp. We denote each such a message $barrier(x)$.

More precisely, a barrier is the guarantee that a group x gives to some other group h that x will not send to h any new messages with a timestamp lower than $barrier(x)$. As all groups receive messages from each other in the order of their timestamps, ensured by fr-mcast, every message exchanged between groups is a barrier.

3.3 Detailed description

To multicast m, p assigns m's initial timestamp and fr-mcasts m to the other processes in g (lines 7–10 in Algorithm 1). Primitive *getTime*() (line 9) returns a unique value based on the process' unique identifier and current wallclock value. After fr-delivering m, each process in g puts m in *messages* (line 13) to be proposed until it is decided (lines 19–20), along with other undecided messages. As all correct processes in g do so, some decision will eventually contain m.

When a set of messages is decided, such messages are handled by each process in g in ascending order of the initial timestamps (line 23). If m has a timestamp that is lower than that of some previously decided message m' (lines 24–25), then $m.ts$ is changed to a value greater than $m'.ts$ (line 26). Doing so will ensure that all messages from g are delivered in the order of their final timestamps.

After deciding m, p checks whether g is one of its destinations (line 27). If so, m is inserted into *stamped* (line 28), meaning that p should deliver it eventually. Then, m is included into *decided* (line 29) so that p stops proposing m in the following consensus instances. As no message decided afterwards within g will have a timestamp lower than m's, p sets $barrier(g)$ to $m.ts$ (line 30). Then, if m has any destination other than g, say h, p fr-mcasts m to h (line 31). When some process q in h fr-delivers m for the first time (lines 11 and 14), q inserts it into *stamped* (line 15) and sets $barrier(g)$ to $m.ts$ (line 16). This is done since any previous message from g has already been received.

Let r be a process in some group $h' \in m.dst$. When (i) the *stamped* set at r contains m, (ii) m has the lowest timestamp among all undelivered messages in such set (lines 32 and 33), and (iii) r has already received a barrier from every group in $sendersTo(h')$ with value greater than $m.ts$ (line 34), r delivers m (lines 35 and 36).

3.4 Liveness guarantees

Assume that group g multicasts m to group h. In Algorithm 1, for process q in h to deliver m, q must receive a barrier $b > m.ts$ from every group k in $sendersTo(h)$. Since the exchange of barriers among processes is triggered by the multicast of application messages, the algorithm assumes that every group in $sendersTo(h)$ periodically multicasts an application message to h. This assumption can be weakened by having each group x in $sendersTo(h)$ multicast a *null message* if x does not have any application messages to multicast to h for a certain timeout Δ. Null messages are not delivered to the application, they only carry a barrier.

Multicasting null messages periodically is a simple solution, however it may introduce unnecessary delays. If q is only waiting for x's barrier to deliver m and x has just sent barrier $b < m.ts$ to h, then unless x has an application message to multicast to h, x will only send another barrier to h after another timeout Δ. As a consequence, the delivery of m may be delayed by up to Δ. Besides, there is no guarantee that the next barrier sent by x to h will be greater than $m.ts$. Another solution is for g, m's source group, to request barriers on behalf of m's destination group h. As soon as process p in g knows the final timestamp of m, p requests a barrier to every x in $sendersTo(h)$. When processes in x receive a barrier request for m, they must run a consensus instance to agree on a barrier with a timestamp greater than m's timestamp. This technique lowers delivery latency at the cost of having more messages exchanged between groups.

Algorithm 1 Baseline Atomic Multicast, for p in group g

```
Initialization
    $k \leftarrow 0$, $messages \leftarrow \emptyset$
    $decided \leftarrow \emptyset$, $stamped \leftarrow \emptyset$, $delivered \leftarrow \emptyset$
    for all $h \in sendersTo(g)$ do
        $barrier(h) \leftarrow 0$

To multicast a message $m$ to groups in $\gamma$
    $m.src \leftarrow g$
    $m.dst \leftarrow \gamma$
    $m.ts \leftarrow getTime()$
    fr-mcast($\{g\}, m$)

when fr-deliver($m$)
    if $g = m.src$ then
        $messages \leftarrow messages \cup \{m\}$
    else if $m \notin stamped$ then
        $stamped \leftarrow stamped \cup \{m\}$
        $barrier(m.src) \leftarrow m.ts$

when $messages \setminus decided \neq \emptyset$
    $k \leftarrow k + 1$
    $undecided \leftarrow messages \setminus decided$
    propose$_g$($k, undecided$)
    wait until decide$_g$($k, msgSet$)
    while $msgSet \setminus decided \neq \emptyset$ do
        let $m$ be in $msgSet \setminus decided$ with smallest timestamp
        let $m'$ be in $decided$ with greatest timestamp, if any
        if $m'$ exists and $m'.ts > m.ts$ then
            $m.ts \leftarrow m'.ts + 1$
        if $g \in m.dst$ then
            $stamped \leftarrow stamped \cup \{m\}$
        $decided \leftarrow decided \cup \{m\}$
        $barrier(g) \leftarrow m.ts$
        fr-mcast($m.dst \setminus \{g\}, m$)

when $stamped \setminus delivered \neq \emptyset$
    let $m$ be in $stamped \setminus delivered$ with smallest timestamp
    if $\forall h \in sendersTo(g) : m.ts < barrier(h)$ then
        deliver($m$)
        $delivered \leftarrow delivered \cup \{m\}$
```

Algorithm variables:

messages: messages multicast by processes in g

decided: messages proposed and decided within g

stamped: messages with a final timestamp sent to g by any group; to be delivered, some messages may need barriers from groups in *sendersTo*(g)

delivered: messages already delivered by p

4 Optimistic Atomic Multicast

In the following, we modify Baseline Atomic Multicast to introduce optimistic delivery. The resulting algorithm, Optimistic Atomic Multicast, can deliver messages optimistically after one communication step, at the risk of delivering some of them out order. In brief, the idea is for a process p to optimistically deliver a message m after p fr-delivers m; to minimize out of order messages, processes may delay the optimistic delivery by a small time window. We first define the Optimistic Atomic Multicast problem (Section 4.1) and then present the new protocol in detail (Section 4.2).

4.1 Problem definition

Optimistic Atomic Multicast is defined by primitives multicast(γ, m), opt-deliver(m) and deliver(m). Hereafter, we refer to the last two primitives as *optimistic* and *conservative* delivery, respectively. Thus, Optimistic Atomic Multicast delivers messages twice, once optimistically and once conservatively, although optimistic delivery requires fewer communication steps than conservative delivery.

Conservative delivery guarantees the five properties of atomic multicast, defined in Section 3.1. Optimistic delivery guarantees validity, uniform integrity, and fifo order of atomic multicast; it ensures non-uniform agreement and probabilistic atomic order (i.e., atomic order is guaranteed under certain assumptions, as defined next).[1]

Applications can exploit this abstraction by processing a message as soon as it is optimistically delivered. If the optimistic delivery order of messages does not match their conservative delivery order, the application may need to rollback the processing of these messages and re-execute them in the correct order (e.g., [14, 15]). However, rolling back the processing of out-of-order messages is not needed if their relative order does not matter to the application (e.g., one message modifies the state of variable x and the other message reads variable y).

4.2 Delivering messages optimistically

In Optimistic Atomic Multicast, processes in the destination set of a message m predict m's atomic order from m's initial timestamp after one communication step. This is done as follows. When a process p in g multicasts m, p fr-mcasts m to all groups in $m.dst$, not only its own group. When q in $m.dst$ fr-delivers m, q optimistically delivers m following the order given by m's initial timestamp.

In order for m's optimistic order to match its conservative order, three conditions are sufficient: (a) there are no message losses; (b) m's source group does not change m's initial timestamp; and (c) before a process in m's destination groups optimistically delivers m, the process receives all messages with initial timestamp smaller than m's.

Condition (a) can be approximated with reliable point-to-point communication (e.g., TCP). We satisfy conditions (b) and (c) probabilistically using the following scheme. Assume p is a process in m's source group or in one of m's destination groups. To satisfy conditions (b) and (c), respectively, p must wait "long enough" before proposing m in a consensus instance and before optimistically delivering m.

The problem is how to define the length of p's *wait window*, denoted $w(p)$. As we consider an asynchronous system, it is impossible to determine $w(p)$ deterministically; instead, we *optimistically assume* that p can estimate $w(p)$. Let $\delta(p,q)$ be the time a message takes to go from q to p and $\epsilon(p,q)$ the skew between the clocks of p and q.[2]

Wait window. For process p in group g, we define $w(p)$ as
$$max_{q \in sendersTo(g)}(\delta(p,q)+\epsilon(p,q))$$

If our optimistic assumption holds, that is, every process can accurately estimate its wait window, then the order of optimistic delivery matches the order of the conservative delivery. In more detail, the optimistic delivery works as follows: After an undecided message m has been fr-delivered by p, p waits until its wallclock has a value greater than $m.ts + w(p)$ and then p opt-delivers m and proposes m in the next consensus instance. At this moment, if the optimistic assumption holds, all messages that could possibly have an initial

[1] Since agreement of optimistic delivery is non-uniform, if the sender crashes while multicasting a message, some processes may deliver the message optimistically but never conservatively.

[2] Notice that a negative skew means that the value of q's clock is higher than the value of p's clock, and so p can wait less for messages from q, as they will have higher timestamps than p's messages.

timestamp lower than $m.ts$ have already been fr-delivered by p. Thus, such messages will be optimistically delivered in the order of their initial timestamps.

If every process q of g has estimated $w(q)$ correctly, then all processes in g will propose messages in the same order, that is, following the initial timestamps. Therefore, there will be no timestamp changes and the optimistic delivery will be correct. This happens because each process q waits for $w(q)$ before proposing a message, and the messages waiting to be proposed are proposed in their initial timestamp order. Not changing timestamps allows for a further improvement: If barriers are requested for each message m, the barrier requests may be sent to the destination groups of m at the same time as m is fr-mcast by its sender. Such requests are also sent with a reliable multicast primitive, before the sender fr-mcasts m in the first step of the algorithm.

If the optimistic assumption fails to hold and some message m is received by some process p after a message m' has been already opt-delivered by p, where $m.ts < m'.ts$, then optimistic delivery violates atomic order. Out-of-order optimistic deliveries are detected by the application since they do not match the order of conservative of deliveries. The application must then handle these out-of-order messages if necessary.

5 Reconfiguration

A reconfiguration allows the *sendersTo* relation to change during the execution. Since including a process in the relation and removing a process from the relation follow similar procedures, we focus next on inclusion. For process p in group g to multicast a message to group h, p creates a reconfiguration request m_{rq}, where $m_{rq}.dst = \{h\}$, timestamps m_{rq} and fr-mcasts it to all processes in g. Upon fr-delivering m_{rq}, it is inserted into *messages*, eventually proposed and decided within g, and then reliably multicast to h. Once processes in h receive m_{rq}, they enqueue a request response m_{rp} to be proposed and eventually decided within h, where $m_{rp}.dst = \{g\}$. As with multicast messages, both m_{rq} and m_{rp} may have their timestamps changed after consensus. The final timestamp of m_{rp} defines the moment from which processes in h will wait for barriers from g. No message m with a timestamp greater than $m_{rp}.ts$ will be delivered by the processes in h until a barrier greater than $m.ts$ is received from g. Once m_{rp} is received by g, it can multicast messages to h. Such messages will have a timestamp greater than $m_{rp}.ts$, to ensure their correct delivery order.

Whenever a message m is sent to h, its sender can request barriers to every group in *sendersTo*(h) so that m can be delivered. For that, all groups in *sendersTo*(h) must know one another. So, as soon as g decides m_{rq}, it fr-mcasts a notification m_{rn} to all groups in *sendersTo*(h), containing the timestamp of the reconfiguration request. When each process $q \in$ *sendersTo*(h) fr-delivers m_{rn}, q knows that it has to send a barrier request to g when sending messages to h with a timestamp greater than $m_{rq}.ts$. This value is chosen because the reconfiguration request itself may be seen as a barrier from g to h. Finally, q checks among the history of messages from its group if there are messages addressed to h with timestamps greater than $m_{rq}.ts$ for which no barrier request has been sent to g. Then a barrier request is sent to g regarding such messages. This is done to ensure progress when a reconfiguration takes place.

6 Implementation

In the following, we comment on the implementation of our prototype and assess the number of communication steps needed for the optimistic and the conservative deliveries. We base our analysis on “good runs", that is, those with no failures, no suspicions of failure, and no reconfigurations. In practice these are (hopefully) the most common cases. In the analysis, we assume a maximum network message delay δ and negligible processing time.

We implemented consensus using Paxos [16]. To ensure termination, Paxos assumes the existence of a leader. In our implementation, one process in each group is assigned the role of leader, coordinating the execution of consensus in its group. To propose a value, a process sends it to the leader of its group, which then proceeds with consensus. The leader optimizes latency by executing Phase 1 of Paxos before a value is proposed [16]. Consensus instances are run in parallel, one for each multicast message. Each group implements an independent instance of Paxos.

To determine $w(p)$, each process p calculates an *estimate delay plus clock skew* for every process q that sends messages to p. The value of $w(p)$ is the maximum among such estimates. Such an estimate, p calculates the average of the difference between its current time and m's timestamp (i.e., $getTime() - m.ts$) for the last 100 messages received from q received by p. Notice that this difference accounts for both $\delta(p,q)$ and $\epsilon(p,q)$.

Consider a message m multicast by p in g to the groups in γ. In order to optimistically deliver m, p fr-mcasts m to all groups in γ and to all the processes in g. This step takes a delay of δ. Instead of delivering m optimistically and starting consensus as soon as m is fr-delivered, processes in g wait until their local time is greater than $m.ts + w(p)$. If process clocks are synchronized and message delay estimations are exact, then $w(p)$ will match δ, and optimistic delivery and the start of consensus will happen with delay δ. If clocks are not synchronized or message delay estimations are inaccurate, then $w(p)$ may be too large, and processes will wait longer than necessary to start consensus, or $w(p)$ may be too small, and consensus may start too early, before a message m' with timestamp smaller than m's is received, leading to the optimistic and conservative delivery of m and m' to happen in different orders.

In the best case, consensus is executed in two communication steps (i.e., 2δ), after which m's final timestamp is determined. Group g must then fr-mcast m to each group h in γ. Since we implement consensus with Paxos and the destinations are learners, sending m's final timestamp to members of h can be done in Paxos' Phase 2B messages, saving one communication step [16]. The optimization works as long as g's leader is not replaced; otherwise, more communication steps are needed [16]. Process q will conservatively deliver m when q receives barriers with timestamp greater than m's final timestamp from every group x in *sendersTo*(h). The barriers necessary for delivering m are requested by p, m's sender, right before fr-mcasting m in the first step of the algorithm. At this time, p only knows m's initial timestamp. Thus, processes in x send a barrier to q that is greater than m's initial timestamp. If m's initial and final timestamps are the same (i.e., the optimistic assumption holds), then q will receive all its needed barriers three communication steps after m is multicast: one step for the request to be transmitted from p to x plus two steps for consensus to be executed in x. In this case, the conservative delivery will have a latency of 3δ. If m's initial and final timestamps differ, a new barrier request will be necessary. In a good run, the timestamp proposed by g's leader is decided, so g's leader can send the new barrier request to x while running consensus in g, leading to a 4δ conservative latency.

7 Performance evaluation

We evaluated the proposed protocols in local-area (Section 7.1) and wide-area networks (Section 7.2). In all experiments, every process multicasts messages to other groups, at a constant rate, according to the *sendersTo* relation. Each run of an experiment took 150 seconds, and latency values are reported as their 95th percentile.

We compare our algorithms with *Paxos amcast*, a protocol we implemented for atomic multicast using Paxos, where a single set of acceptors act as a fault-tolerant sequencer for all messages multicast in the system, providing a global order for them.

7.1 Local-area network experiments

The local-area network experiments were executed in a cluster of HP SE1102 servers, each with two quad-core Intel Xeon L5420 processors and 8GB of main memory, running CentOS 6.2 64 bits. The servers were interconnected through an HP ProCurve2910al-48G Gigabit switch and their clocks were kept approximately synchronized by using NTP. There were five groups, each with three processes; processes were distributed among five different servers of the cluster.

In the following, we evaluate how our protocols behave with respect to different loads (Section 7.1.1), different numbers of groups multicasting to each other (Section 7.1.2), and different estimations of the wait window for the optimistic delivery (Section 7.1.3).

7.1.1 Throughput, latency and mistakes versus load

Figure 1 shows how throughput and latency vary with the load, measured in number of messages multicast per process per second. The maximum throughput of each protocol is circled in the graph. We define maximum throughput as the throughput value that corresponds to the latency before its inflection point (also marked in the graph). Optimistic Atomic Multicast's maximum throughput is determined by the conservative delivery.

We can see that the maximum throughput of Optimistic Atomic Multicast (graph on the left) is lower than the baseline's. This is due to the higher overhead of Optimistic Atomic Multicast (i.e., the initial reliable

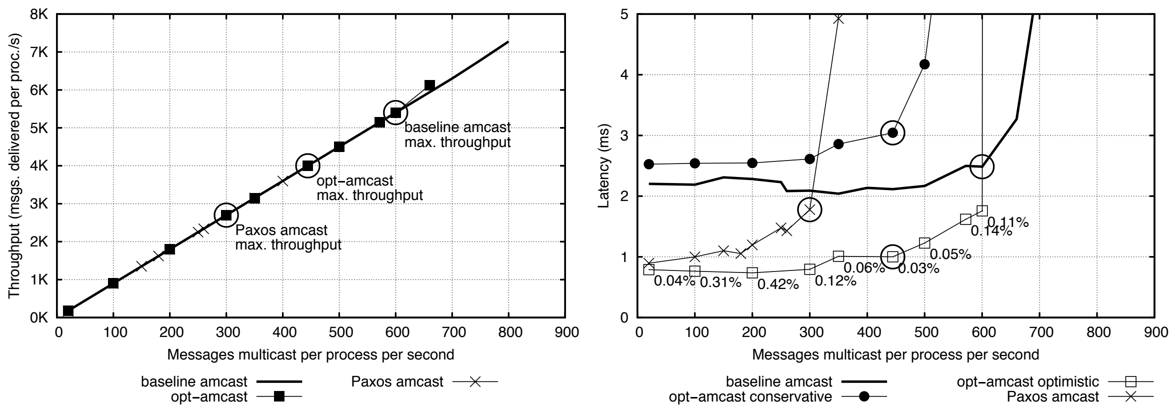

Figure 1: Throughput, latency and percentage of mistakes for $|sendersTo(g)| = 3$, for every g, in a local-area network.

multicast is sent to all destination groups, as opposed to the sender's group only). Paxos amcast has the lowest maximum throughput, since all messages are handled by the same set of acceptors.

The latency of optimistic delivery at the protocol's maximum throughput (graph on the right) is significantly lower than that of Paxos amcast and less than half that of baseline and conservative. The conservative delivery latency is higher than baseline's because, in the former, messages are proposed for consensus only after being opt-delivered, whereas in the latter they are proposed as soon as they are fr-delivered. Among the non-optimistic deliveries, Paxos amcast achieves the lowest latency since it does not have to send or wait for barriers to deliver messages.

Figure 1 (right) also shows the percentages of mistakes of the optimistic delivery, i.e., the percentage of messages whose optimistic order does not match the conservative order. By using NTP to synchronize clocks and estimating a wait window before opt-delivering messages, the mistakes rate remained below 0.5%.

7.1.2 Impact of the number of communicating groups

From Figure 2 (top left), the maximum throughput of each protocol tends to increase with the number of destinations of each message. This happens because as the number of destinations increases, more groups will deliver a message decided by each consensus instance. Therefore, configurations with more destinations make better use of consensus executions. As the number of destinations for each message augments, the throughput difference between Paxos and the other algorithms decreases; when every message is addressed to every group, all acceptors notify all processes in the system, which is the normal case for Paxos amcast, but not for the baseline and Optimistic Atomic Multicast. Even in such situation, Paxos amcast has lower throughput, since it has only one set of acceptors ordering all messages, while the other protocols have one set of acceptors per each group, distributing the load.

Still in Figure 2 (bottom left), we can see that the number of communicating groups has an impact on the optimistic delivery latency. This happens because the optimistic delivery latency in each process p is affected by the wait window $w(p)$, which is based on the worst latency among the processes sending to p; as more processes are included in the calculation of $w(p)$, the chances of getting a process with a higher estimated delay plus clock skew increase. A more significant effect can be noticed on the baseline and conservative latencies. This happens because not always all required barriers are received immediately when a message is enqueued for delivery.

7.1.3 Impact of the estimated delay and clock skew

Figure 2 (top right) shows the latency and percentage of mistakes of the optimistic delivery on maximum throughput when each group can send messages to three other groups (i.e., $\forall g : |sendersTo(g)| = 3$). We can see that both the conservative and the optimistic delivery latencies increase with the length of the wait window, which depends on the estimated delay δ and clock skew ϵ. The conservative delivery increases

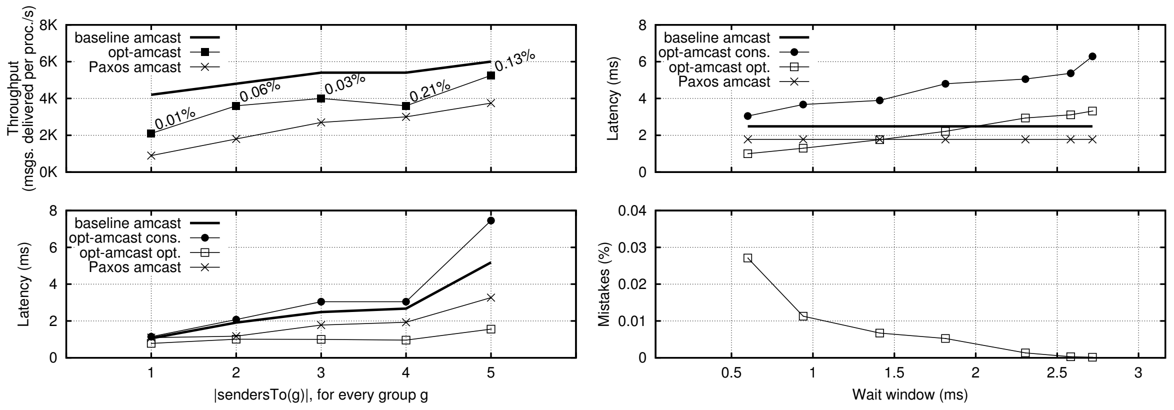

Figure 2: Impact of number of communicating groups and wait-window size on throughput and latency (percentages are mistakes in optimistic delivery).

because a message is proposed only after it is opt-delivered, and so, the final delivery order tends to be the same as the optimistic one. As a reference, we show the latencies of the baseline protocol and Paxos amcast.

The percentage of mistakes decreases as the wait window becomes larger, as depicted in Figure 2 (bottom right). Waiting for more than 1 ms proved not to be very effective in comparison to how much it increased latency. Still, such small increase of the wait window (from 0.6ms to 0.9ms, approximetely) was enough to reduce the mistakes rate by more than half, leading to an accuracy of over 99.98%.

7.2 Wide-area network experiments

The wide-area experiments used four m1.large Amazon EC2 instances[3] running Ubuntu Server 12.04.1 64 bits. Each instance was located in a different geographical region: three in the United States (Virginia, California and Oregon) and one in Ireland. To synchronize clocks, each of such instances queried a nearby public GPS-based NTP server, resulting in synchronization offsets below 10ms. It is worth noting that this deployment was fairly heterogeneous, since instances from different regions had different levels of processing power and network bandwidth. To report maximum throughput, we chose the load up to which latency was stable in every process. There were four multicast groups, each with four processes, one process from each geographical region.

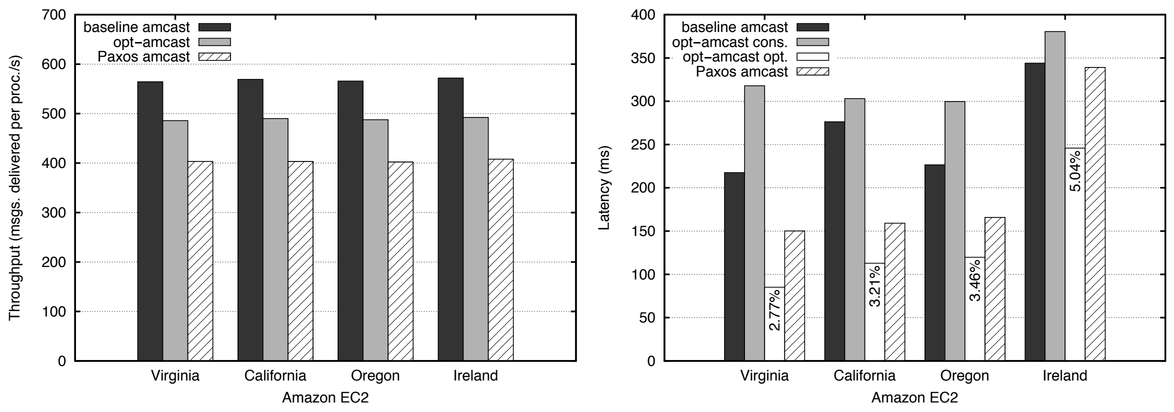

Figure 3: Throughput, latency and percentage of mistakes for $|sendersTo(g)| = 2$, for every g, in a wide-area network.

In Figure 3, we can see the throughput and latency results in a WAN. As in the cluster, the maximum

[3]http://aws.amazon.com/ec2

throughput achieved by baseline and Optimistic Atomic Multicast are fairly higher than that of Paxos amcast, with baseline's as the highest. Regarding latency, the optimistic delivery took the shortest time, in every region. We can see a high latency variation depending on the geographic location. The processes in Ireland experienced the highest latency, for every protocol evaluated. On the other extreme, the processes in Virginia had the lowest latencies for most protocols.

Finally, the rate of mistakes in the WAN experiments was much higher than in the cluster. It is a known fact that latencies in the Internet are much less predictable than in a local-area network, so a lower accuracy was not surprising. Still, we managed to achieve an accuracy around 95%. This means that, even if all messages sent to a given destination require strict ordering among them, only 5% might cause rollbacks. However, many applications can tolerate out-of-order messages (e.g., two messages with requests that access different rows in the same partition). Therefore, when accounting for application semantics, 5% of out-of-order messages will likely result in less than 5% of rollbacks.

8 Related work

There is a plethora of works in the literature related to atomic broadcast and multicast [17]. We review here those which are the most relevant to our work. To the best of our knowledge, no previous work presents all the features provided by our protocol, which is a quasi-genuine, reconfigurable, fault-tolerant, fifo atomic multicast that can deliver messages in three communication steps, with an optimistic delivery within one single step.

8.1 Atomic broadcast and multicast

Our atomic multicast protocols order messages using a mechanism that resembles Lamport's total order algorithm [1]: analogously to our barrier mechanism, a process considers an event to be the next in a total order of events when the process knows that such event has the lowest timestamp among all possible events to be considered in the system. Similarly to our protocols, Lamport's total order algorithm also relies on fifo, reliable channels. Differently than our protocols, it requires the participation of all processes in the system and does not tolerate failures.

One could see our barrier mechanism as a deterministic merge [18] algorithm. In particular, our protocols model each group as a merger and producer at the same time. To ensure that all mergers receive the same set of messages of all producers, we employ consensus and fifo reliable multicast. Finally, the merging is done by sorting all undelivered messages according to their timestamps and delivering those which have a timestamp up to the lowest of the last barriers received from the groups in *sendersTo*. Differently from [18], however, our protocols tolerate failures and do not require all mergers to receive all messages from all producers. Instead, each group may have a different *sendersTo* set. We also support reconfiguration, allowing the *sendersTo* relation to change over time.

Two fault-tolerant genuine atomic multicast algorithms are presented in [11]. Both are based on Skeen's multicast algorithm [19]. In the first algorithm, the destinations of each multicast message exchange timestamps for the message and, once each destination has received a timestamp from a majority of processes from each destination group, consensus is run among the destinations to decide the final timestamp. Upon decision, the message destinations coordinate to determine the message's position in the delivery queue. Such a protocol can deliver messages in 4δ, assuming the best-case bound of 2δ for consensus. A variation of this algorithm [20], optimized for message size, has delivery latency of 5δ.

The second algorithm in [11], similarly to [13] and [21], has an optimal delivery latency of two inter-group communication delays. Although based on Skeen's algorithm, they can tolerate failures by replacing each process by a group of processes. To deliver messages with low latency, communication within each group must be fast, so each group is contained in a single site (e.g., the same local-area network). As a result, these protocols are vulnerable to disasters, that is, the failure of the site. Although messages can be delivered with two inter-group delays, overall delivery latency (i.e., considering inter group and intra group communication) is 6δ in most cases and 4δ in a few special cases.

In [13], the authors propose a broadcast protocol which delivers messages in atomic order within one single communication step. This protocol is based on rounds: each process delivers the received messages in a round only after receiving the messages from all other groups for that round. To ensure liveness, groups need to constantly exchange messages, even if null. Rounds are conceptually similar to our barrier mechanism. Since our multicast protocols are quasi-genuine, not all groups have to coordinate to deliver messages.

Furthermore, the *sendersTo* relation can be reconfigured at any time based on the communication patterns of the application.

8.2 Optimistic total order

Optimistic techniques have been used before to reduce the latency of atomic broadcast. To our knowledge, the protocol proposed in this technical report is the first to exploit optimism in atomic multicast. Moreover, differently from previous approaches, our optimistic assumption does not rely on spontaneous total order, a property that typically holds in local-area networks under moderate load, but not necessarily in geographically distributed systems.

Two optimistic atomic broadcast protocols are proposed in [5] and [7]. In both cases, if the optimistic assumption holds, messages are delivered within two communication delays. The idea of the algorithm in [5] is to rely on spontaneous total order to avoid ordering through consensus messages that have already been spontaneously ordered by the network. Fast Paxos [7] is an extension of the classic Paxos algorithm. Differently from classic Paxos, messages are sent directly to acceptors (skipping the leader). If messages are received in the same order, delivery is done in two communication steps; otherwise, it takes longer.

Broadcast algorithms that take into consideration application semantics to deliver messages fast have been also proposed [8, 9, 10]. To a certain extent, these algorithms "optimistically" assume that not every two messages must be delivered in the same order by all processes, but only some messages (e.g., messages that access shared objects). Optimistic Atomic Multicast does not rely on application semantics. We note however that application semantics could be used when deciding whether the order of optimistic and conservative delivery match.

In [6], the authors propose a technique that approximates spontaneous ordering in a wide-area setting. The key idea is for every process to insert artificial delays in incoming messages. The resulting delay between each process p and every other process should then become the same as the delay between p and a given sequencer s (e.g., the leader in a consensus-based protocol). For the technique to work properly, the communication latency between each pair of processes has to be approximately constant. Our optimistic delivery does not have one single sequencer process as a reference, as opposed to [6]. Instead, given a set of processes which communicate with one another, each one of such processes is able to determine an optimistic ordering based solely on its own wait window and on messages' timestamps. Furthermore, the authors propose an atomic broadcast protocol,[4] where the latency between every single pair or processes must be constant, whereas we propose an optimistic atomic multicast protocol which cares only about those pairs of processes that can communicate, as defined by the *sendersTo* relation.

9 Conclusion

This work introduced the concept of quasi-genuine multicast, a class of protocols based on pre-determined communication patterns (i.e., the *sendersTo* relation). Such patterns are based on the application and may be changed on-the-fly (i.e., by means of a system reconfiguration). Experimental results in both local-area and wide-area networks show that Optimistic Atomic Multicast is effective in increasing throughput and reducing latency. Low latency is due to a probabilistic assumption that was verified in 95% of deliveries in a wide-area network and over 99.5% in a local-area network. As a future work, we intend to investigate protocols that automatically detect communication patterns and reconfigure the system as pairs of groups communicate more or less frequently.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[2] F. B. Schneider, "What good are models and what models are good?," in *Distributed Systems* (S. Mullender, ed.), ch. 2, Addison-Wesley, 2nd ed., 1993.

[3] V. Hadzilacos and S. Toueg, *Fault-tolerant broadcasts and related problems*, pp. 97–145. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993.

[4] L. Lamport, "Lower bounds for asynchronous consensus," *Distributed Computing*, vol. 19, no. 2, pp. 104–125, 2006.

[4] In [6], the protocol is called "atomic multicast", but is defined as a protocol in which every correct process delivers all messages multicast by all correct processes. From our definitions, we call it "atomic broadcast".

[5] F. Pedone and A. Schiper, "Optimistic atomic broadcast: a pragmatic viewpoint," *Theoretical Computer Science*, vol. 291, no. 1, pp. 79 – 101, 2003.

[6] A. Sousa, J. Pereira, F. Moura, and R. Oliveira, "Optimistic total order in wide area networks," in *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, SRDS '02, pp. 190–199, IEEE Computer Society, 2002.

[7] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[8] F. Pedone and A. Schiper, "Generic broadcast," in *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99, formerly WDAG)*, 1999.

[9] L. Lamport, "Generalized consensus and paxos," Tech. Rep. MSR-TR-2005-33, Microsoft Research (MSR), Mar. 2005.

[10] P. Sutra and M. Shapiro, "Fast Genuine Generalized Consensus," in *Symposium on Reliable Distributed Systems*, (Madrid, Spain), Oct. 2011.

[11] R. Guerraoui and A. Schiper, "Genuine atomic multicast in asynchronous systems," Tech. Rep. 98/273, EPFL, Mar. 1998.

[12] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proc. VLDB Endow.*

[13] N. Schiper and F. Pedone, "On the inherent cost of atomic broadcast and multicast in wide area networks," in *Proceedings of the 9th international conference on Distributed computing and networking*, ICDCN'08, 2008.

[14] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 4, pp. 1018–1032, 2003.

[15] F. R. Cecin, C. F. R. Geyer, S. Rabello, and J. L. V. Barbosa, "A peer-to-peer simulation technique for instanced massively multiplayer games," in *Proceedings of the 10th IEEE international symposium on Distributed Simulation and Real-Time Applications*, 2006.

[16] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[17] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, pp. 372–421, December 2004.

[18] M. K. Aguilera and R. E. Strom, "Efficient atomic broadcast using deterministic merge," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, (New York, NY, USA), pp. 209–218, ACM, 2000.

[19] K. Birman and T. Joseph, "Reliable communication in the presence of failures," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 1, pp. 47–76, 1987.

[20] L. Rodrigues, R. Guerraoui, and A. Schiper, "Scalable atomic multicast," in *Computer Communications and Networks, 1998. Proceedings. 7th International Conference on*, pp. 840 –847, oct 1998.

[21] J. Fritzke, U., P. Ingels, A. Mostefaoui, and M. Raynal, "Fault-tolerant total order multicast to asynchronous groups," in *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pp. 228–234, 1998.

Appendix: Proof of correctness

Here, we prove that the algorithm ensures the properties defined in section 3.1, i.e., validity, uniform agreement, uniform integrity, atomic order and fifo order. To do this, we prove that Algorithm 1 ensures these properties, as long as groups keep sending barriers periodically to each other—this is trivial to guarantee by periodically multicasting null messages, which are obviously never delivered to the application: every process p, of each group g, periodically multicasts null messages to every group $h : g \in \mathit{sendersTo}(h)$. We are also assuming that the majority of processes of each group is correct, and that there is no concurrency on the execution of the algorithm in any single process.

Lemma 1. *Once every correct process in a group g has inserted a message m into messages, m will eventually be decided by all correct processes of g.*

Proof. In Algorithm 1, once each correct process p from g has received m and inserted into its *messages* set, m will be proposed by p in every consensus instance, until m has been inserted into *decided* (lines 19 and 20). As m is inserted into *decided* only after being decided (l. 29), every correct process of g will propose m at some point. Since eventually every faulty process crashes, then only correct processes remain at some point, so m will eventually be proposed by all remaining processes and will, therefore, be decided. From the uniform agreement property of consensus, as m is decided by some process of g, all correct processes of g decide m. □

Lemma 2. *Once a message m has been multicast by a correct process p from group g, m will eventually be decided in g.*

Proof. In Algorithm 1, whenever a correct process in a group g multicasts a message, it is first fr-mcast to all processes in g (l. 10). From the properties of the fr-mcast primitive, every correct process from g fr-delivers m and inserts it into *messages* (l. 13). From Lemma 1, m will eventually be decided within g. □

Lemma 3. *Once a message m has been decided in its source group g, m will be received by all correct processes to which m has been addressed and then m will be inserted into their respective stamped sets.*

Proof. When a message is decided in its source group, if $g \in m.\mathit{dst}$, every correct process of g also inserts m into *stamped* (l. 28). Besides, each correct process in g fr-mcasts m to all other groups that m is adressed to (l. 31).

Let q be any correct process in a destination group h of the message m, such that $h \neq g$. From the properties of fr-mcast and the fact that at least one correct process in g fr-mcasts m (again, we are assuming that at least a majority of processes is correct in each group), q will fr-deliver m. Once m is fr-delivered by q, such message is inserted into the *stamped* set of q, unless this has been already done (lines 14 and 15). □

Lemma 4. *Given a correct process p from group g, and a message m, p eventually receives some message $m' : m'.ts > m.ts$ from every group in $\mathit{sendersTo}(g)$.*

Proof. Every process, in every group h, multicasts null messages periodically, with monotonically increasing timestamps, to every group $i : h \in \mathit{sendersTo}(i)$. Therefore, all groups in $\mathit{sendersTo}(g)$ will eventually multicast null messages with a timestamp higher than $m.ts$. From Lemma 2 and Lemma 3, all such null messages will be received by p.

□

Lemma 5. *Given any two messages m and m', multicast by processes in the same group g, if g decides that their final timestamps are such that $m.ts < m'.ts$, then no process $p \in g$ fr-mcasts m' to any group $h \neq g$ before m.*

Proof. Assume, by way of contradiction, that p fr-mcasts m' first. From lines 21 to 31, this means that m' was decided first, so the final timestamp of m can only be greater than that of m', a contradiction. □

Lemma 6. *Given any two processes p and q and a message m, if p and q deliver m, then p and q agree on the final timestamp of m.*

Proof. Let g be the source group of m, $m.src$. We must prove that every process y that delivers m (including p and q) agrees on the final timestamp of m with every correct process of g. Let us consider the case where y belongs to g. In that case, as y delivers m, then y decided it previously. From the algorithm, consensus instances are decided by each process in the order of their identifiers. As y delivers m, y has decided each consensus instances up to k_m, where k_m is the consensus intance in which m is decided. We first prove that, if $y \in g$, then y agrees with every correct process of g regarding the final timestamp of m. Such proof can be done by induction on the identifier k of each consensus instance within g. Let r be any correct process of g:

Base case ($k = 1$): As this is the first consensus instance within g, it means that *decided* is empty when such instance started. From the uniform agreement property of consensus, we know that y and r decide the same contents for *msgSet* (l. 21). Each message included in such agreed set also includes an initial timestamp field. As the *decided* set is empty, such timestamps are not changed in neither y nor r (lines 23 to 26). Therefore, both processes consider the same timestamp value for every message. Besides, as the *msgSet* is the same for both, the *decided* set remains identical in y and r.

Induction step: Suppose that y and r have already learnt the decisions of instance k, their *decided* sets remained identical and they decided the same timestamp value for each message sent from some process in their group so far. From the algorithm, all processes within a group learn all decisions in the same order, which is that of the consensus instance identifiers. Therefore, both y and r will next learn the decision of instance $k+1$ (at least if $k+1 \leq k_m$, since we know that y decides k_m, but we do not know if y is correct). From the uniform agreement property of consensus, the *msgSet* decided is the same for both processes. In Algorithm 1, the timestamps may be changed after the consensus decision only in line 26, based on what is already in the *decided* set (lines 23 to 26). As the *msgSet* and *decided* sets are each identical in y and r, they will make the exact same change to the timestamp of each message in the *while* loop (lines 23 to 31). Therefore, they also agree on the timestamps of each message decided on consensus instance $k+1$ (up to k_m) and their *decided* sets remain identical.

Therefore, if $y \in g$, y agrees with every correct process of g regarding the final timestamp of m. Now we prove that this happens also in the case where y belongs to group $h \neq g$, where h is a destination of m. Let r be any correct process of g. After setting the final timestamp of m (l. 23 to l. 31), r fr-mcasts m to h (l. 31). From the algorithm, after fr-delivering m, y never changes the value of $m.ts$, set by r in accordance with all correct processes of g (from the uniform agreement property of consensus).

□

Lemma 7. *If a process p from a group g has received a message m from another group $h \neq g$, then p has also received from h any message $m' : m'.ts < m.ts$, where $m.ts$ and $m'.ts$ are final timestamps.*

Proof. Let us assume, by means of contradiction, that p has received m before m'. It means that some process $q \in h$ has fr-mcast m' before m. As a group receives any message from another group only after such message's final timestamp has been decided, then either the final timestamp of m has not been decided yet, or it has been decided and q has not fr-mcast it to g. In the former case, since m' is already in the *decided* set of q, the final timestamp of m will be set to be $m.ts > m'.ts$ before being fr-mcast to g, which is a contradiction. The latter case, i.e., m had already been decided by q, but not fr-mcast yet, is also a contradiction, from Lemma 5.

□

Lemma 8. *Once a process p has inserted a message m into its stamped set, no message $m' : m'.ts < m.ts$ from $m.src$ will be inserted into such stamped set afterwards.*

Proof. Let g be the group of p. Group g is either $m.src$ or not. In the former case, before inserting any message $m' : m'.src = m.src$ into *stamped*, p has to decide m' first. When a message is decided by p, p checks whether some other message which was decided previously has a timestamp lower than m' and, if that is the case, the timestamp of m' is changed to a value greater than that of any other message already decided (lines 23 to 26 of Algorithm 1). Only then m' may be inserted into the *stamped* set of p (l. 28). Therefore, p inserts messages from other processes of g into its *stamped* set in ascending order of timestamps. Besides, the messages from g are fr-mcast to other groups in this same order (l. 31) by all correct processes of g, from the uniform agreement property of consensus.

In the case where $g \neq m.src$, p inserts m into *stamped* in line 15. Considering that when p receives any message from $m.src$, any other message from $m.src$ with a lower timestamp has already been received (from Lemma 7), we can infer that, once p inserts m into *stamped*, no message $m' : m'.ts < m.ts$ from $m.src$ will be inserted into the the *stamped* set of p afterwards. □

Lemma 9. *Once a message $m \neq null$ has been inserted into the stamped set of a correct process p from group g, m is eventually delivered by p.*

Proof. Eventually, p will have received, from every group in *sendersTo*(g), some message with a timestamp greater than $m.ts$ (from Lemma 4), i.e., at some point all barriers required for p to deliver m will have been received. Once this happens, no more messages with a timestamp lower than that of m will be inserted into the *stamped* set of p (from the definition of the *sendersTo* relation and from Lemma 8). Let *ready* be the set of m plus all undelivered messages from *stamped* whose timestamps are lower than that of m, that is, $ready = \{m\} \cup \{m' : m' \in stamped \setminus delivered \wedge m'.ts < m.ts\}$.

The barriers received by p allow the delivery of all messages in *ready*: such barriers are all greater than the timestamp of m, and all other messages in *ready* have timestamps lower than $m.ts$. Each one of these messages will eventually be inserted into *delivered* and, if it is different from *null*, it will be delivered by p. We prove this by induction on the position i of each message m_i in *ready*, in ascending order of timestamps.

Base case ($i = 1$): Let m_1 be the first message in *ready*, i.e., $\nexists m \in stamped \setminus delivered : m.ts < m_1.ts$. We know that all the necessary barriers have been received already, so m_1 satisfies all conditions from lines 33 and 34 of Algorithm 1. Therefore, m_1 will be inserted into *delivered*, after which m_1 will no longer belong to $stamped \setminus delivered$. Also, if $m_1 \neq null$, m_1 will be delivered.

Induction step: Suppose that m_i will eventually be inserted into the *delivered* set. Once this happens, it means that m_i was the first message in $stamped \setminus delivered$ in ascending timestamp order. Since no more messages have been inserted into *ready*, as soon as m_i is inserted into *delivered*, m_{i+1} will be the first one in $stamped \setminus delivered$, having, therefore, the lowest timestamp in such set. Then, as all barriers necessary for m have already been received and $m_{i+1}.ts \leq m.ts$, m will satisfy both conditions of lines 33 and 34 of Algorithm 1. Thus, m_{i+1} will also be inserted into the *delivered* set and, if $m_{i+1} \neq null$, it will be delivered. □

Proposition 1. (VALIDITY).
If a correct process p multicasts a message m, then every correct process q that is a destination of m delivers m.

Proof. Immediate from Lemma 2, Lemma 3 and Lemma 9. □

Proposition 2. (UNIFORM INTEGRITY).
For any message m, every process p that is a destination of m delivers m at most once, and only if some process has multicast m previously.

Proof. In other words, this property means: If a message m was delivered by some process p of group g, then (1) m has been multicast before, (2) $g \in m.dst$ and (3) it has not been delivered by p before. From lines 33 to 36 of Algorithm 1, and since no message is removed from *delivered*, no message can be delivered twice, satisfying (3). For a message m to be delivered (l. 35), it must belong to the *stamped* set (l. 33). There are two possibilities to when m has been inserted into such set:

- m has been originated in the same same group of p, that is, $p \in m.src$, which means that m was inserted into *stamped* in line 28 of Algorithm 1, or
- m has been sent from a group $h \neq g$, which means that it was inserted by p into *stamped* in line 15.

For the case when $p \in m.src$, m can be inserted into *stamped* only in line 28, which means that $g \in m.dst$, satisfying condition (2) for this case. Also, this happens only when m has been decided in some consensus instance within g. To have been decided within g, from the properties of consensus, we know that it must have been proposed by some process of g, which happens in line 20. In line 20, for a message to be proposed, it must be in $messages \setminus decided$, as the contents of such set are the value proposed. For a message to be in *messages*, it must have been inserted there, which happens only in line 13. For l. 13 to be executed, m must have been fr-delivered and g must be the source group of m, that is, $g = m.src$. For a message to be fr-mcast by a process to its own group, a multicast(m) call must have been made—which satisfies condition (1)—, for line 10 is the only one where a process fr-mcasts a message to its own group.

As for the case when $p \notin m.src$, m is inserted into the $stamped$ set of process p in line 15, when it has been fr-delivered. For a process $q \in h$, where $h = m.src$, to fr-mcast m, m must have been decided within h, which means that it was proposed within h. Therefore it was multicast—satisfying condition (1)—by h to g. From line 31, g necessarily belongs to $m.dst$, satisfying condition (2). □

Proposition 3. (UNIFORM AGREEMENT).
If a process p delivers a message m, then every correct process that is a destination of m delivers m.

Proof. Let g be the group of p. For m to be delivered by p, m has to belong to the *stamped* set of p. To be included in such set, there are two possibilities: either m was decided in g, being inserted in *stamped* in line 28 of Algorithm 1, or m was decided in some other group h, fr-mcast by some process $q \in h$, fr-delivered by p and inserted into *stamped* in line 15. Either way, m was decided at some point. From Lemma 3, all correct processes that are destinations of m insert m into their respective *stamped* sets as well. Finally, from Lemma 9, all correct processes that are destinations of m eventually deliver m. □

Lemma 10. *If two messages m and m', both different from null, have been multicast to group g, and their final timestamps are such that $m.ts < m'.ts$, then no process $p \in g$ delivers m' before m.*

Proof. Suppose, by way of contradiction, that m' is delivered by p before m. Message m either has already been inserted into *stamped* or not. In the former case, as $m \neq null$ and m has not been delivered, then m belongs to $stamped \setminus delivered$. Therefore, m' could not have been delivered, since $m.ts < m'.ts$ and it would not satisfy the condition from line 33 until m has been delivered, so we have a contradiction in the case where m was already in *stamped*.

The other case is when m has not been inserted into *stamped*. From line 34 of Algorithm 1, and since m' has been delivered, we know that p has received some barrier $b > m'.ts$ from every group in $sendersTo(group(p))$. Therefore, from Lemma 8, and since any barrier is also a message, we know that any new message that arrives at p will have a timestamp greater than $m'.ts$. As m has arrived after m', then $m.ts > m'.ts$, which is also a contradiction. □

Lemma 11. *Consider a precedence relation, denoted by "$\rightarrow$", between delivered messages such that (a) such relation is acyclic and (b) if $m \rightarrow m'$, then no process in both γ and γ' (destination sets of m and m', respectively) deliver m' before m. If there is any process in both γ and γ', then either $m \rightarrow m'$ or $m' \rightarrow m$.*

Proof. In Algorithm 1, the relation denoted by "$\rightarrow$" may be defined as the order of the messages' (unique) final timestamps: $m.ts < m'.ts \implies m \rightarrow m'$, where $m.ts$ and $m'.ts$ are final timestamps. As every message has a unique timestamp, then either $m.ts < m'.ts$ or $m'.ts < m.ts$, for every two messages m and m'. The timestamp order is acyclic. Assume, by means of contradiction, that it is not acyclic; then it would be possible to have $m.ts < m'.ts$ and $m'.ts < m.ts$, which is a contradiction.

The timestamp order is stronger than the "$\rightarrow$" relation, since the former applies to every pair of messages. Therefore, we can use it to prove that this lemma is true, as follows. From Lemma 6, all destinations of m and m' agree on their final timestamps before delivering it (i.e., either $m.ts < m'.ts$ or $m'.ts < m.ts$). From Lemma 10, if $m.ts < m'.ts$, then no process in a group that is a destination of both m and m' delivers m' before m. □

Proposition 4. (ATOMIC ORDER).
No two processes p and q in both γ and γ' (destination sets of m and m', respectively) deliver m and m' in different orders.

Proof. We prove that Lemma 11 implies the atomic order property. Assume, by way of contradiction, that Lemma 11 holds for Algorithm 1, while the atomic order property does not. From Lemma 11, if both m and m' are delivered, then either $m \rightarrow m'$ or $m' \rightarrow m$, and no process violates such precedence relation. However, as the atomic order does not hold, p and q may deliver m and m' in different orders, violating the precedence relation "$\rightarrow$", which is a contradiction. □

Lemma 12. *If a process p from group g multicasts m' after m, then the final values of their timestamps will be such that $m.ts < m'.ts$.*

Proof. As both messages are sent from g, they have to be decided within g. If m and m' are decided in different consensus instances, and m is decided first, then, from lines 23 to 26, m' will necessarily have a timestamp greater than that of m. Therefore, the two possible ways of having $m'.ts < m.ts$ are: either m' is decided before m, or both are decided in the same consensus instance, but the timestamp of m is set to a value higher than that of m'.

From lines 9 and 10, and the properties of fr-mcast, we know that all correct processes of g fr-deliver m, then m', which means that each process q from g also inserts m, then m' into its *messages* set. Suppose, by way of contradiction, that m' is decided before m. This implies that some process $q \in g$ proposed a message set that included m', but not m. As q inserted m before m' into *messages*, the only possible way of proposing m' and not m is by already having decided m previously, which is a contradiction. Therefore, m' is not decided before m.

As for the case where m and m' are decided in the same consensus instance, we know, from line 9, that the initial timestamp of m is already smaller than that of m'. Therefore, from line 23, m is inserted into *decided* first. Even if the timestamp of m has changed, as m was already in *decided* before m', we know that the final timestamp of m' will again be made greater than that of m (from lines 23 to 26). □

Proposition 5. (FIFO ORDER).
If a process p multicasts m and then m' to γ and γ', respectively, then no process q in both γ and γ' delivers m' before delivering m.

Proof. Immediate from Lemma 10 and Lemma 12. □