Università della Svizzera italiana

Faculty of Informatics

USI Technical Report Series in Informatics

Fast Atomic Multicast

Paulo R. Coelho[1], Nicolas Schiper[2], Fernando Pedone[1]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland
[2] École Polytechnique Fédérale de Lausanne, Switzerland

Abstract

Atomic multicast is a communication building block of scalable and highly available applications. With atomic multicast, messages can be ordered and reliably propagated to one or more groups of server processes. Because each message can be multicast to a different set of destinations, distributed message ordering is challenging. Some atomic multicast protocols address this challenge by ordering all messages using a fixed group of processes, regardless of the destination of the messages. To be efficient, however, an atomic multicast protocol must be genuine: only the message sender and destination groups should communicate to order a message. We present FastCast, a genuine atomic multicast algorithm that offers unprecedented low time complexity, measured in communication delays. FastCast can order messages addressed to multiple groups in four communication delays; messages addressed to a single group take three communication delays. In addition to proposing a novel atomic multicast protocol, we extensively assess its performance experimentally.

Report Info

Published
March 2017

Number
USI-INF-TR-2017-2

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

Many modern online applications require scalable performance and high availability. Designing systems that combine scalability and fault tolerance, however, is challenging. Some systems respond to the challenge by weakening the guarantees they offer to the clients (i.e., weak consistency). While weak consistency has proved successful in some contexts (e.g., [3, 7, 9, 30]), it is not appropriate to every application and often places the burden on the clients, who must cope with non-intuitive application behavior. Strong consistency (e.g., linearizability [16]) leads to more intuitive application behavior but requires client requests to be ordered across the system before they are executed by the servers [18, 28].

Atomic multicast is a communication building block that allows messages to be propagated to groups of processes with reliability and order guarantees. Intuitively, all non-faulty processes addressed by a message must deliver the message and processes must agree on the order of delivered messages. Atomic multicast offers strong communication guarantees and should not be confused with network-level communication primitives (e.g., IP-multicast), which offer “best-effort" guarantees. Because messages can be multicast to different sets of destinations and interleave in non-obvious ways, implementing message order in a distributed setting is challenging. Some atomic multicast protocols address this challenge by ordering all messages using a fixed group of processes or involving all groups, regardless of the destination of the messages. To be efficient, however, an atomic multicast algorithm must be *genuine*: only the message sender and destination processes should communicate to propagate and order a multicast message [14]. A genuine atomic multicast is the foundation of scalable systems, since it does not depend on a fixed group of processes and does not involve all processes.

In this paper, we introduce FastCast, a genuine atomic multicast algorithm that offers unprecedented low time complexity, measured in communication delays. FastCast can order and deliver *global messages* (i.e., messages addressed to multiple groups of processes) in four communication delays; *local messages* (i.e., messages addressed to a single group of processes) take three delays. In comparison, three communication delays is a lower bound on atomic broadcast (in the presence of collisions) [20], a communication abstraction where there is a single group of processes.

FastCast is an optimistic algorithm inspired by BaseCast, an earlier genuine atomic multicast algorithm that requires six communication delays [5, 13, 25]. FastCast's secret sauce is to decompose the procedure used by BaseCast to order global messages in two execution paths. There is a *fast path* that speculates about the order of messages and a *slow path*, similar to BaseCast. If the fast path's "guess" is correct, something that can be assessed after four communication delays, the slow path is abbreviated; otherwise the slow path continues and computes the final order of the message. FastCast provides a significant advantage to existing atomic multicast algorithms since the fast path is correct in most common cases, that is, when the message sender and destination processes do not fail and are not suspected to have failed.

In addition to proposing a novel atomic multicast algorithm with reduced number of communication delays to order global messages, we have fully implemented FastCast and compared its performance to BaseCast and a non-genuine atomic multicast protocol. We conducted experiments in three environments: a local-area network (LAN), an emulated wide-area network (emulated WAN), and a real wide-area network (WAN). We evaluated FastCast with a microbenchmark with configurations involving different number of groups, up to 16 groups, and with a social network application deployed in 48 servers distributed in 16 groups of 3 servers each.

In brief, our results show that in WAN environments, FastCast outperforms BaseCast and the non-genuine atomic multicast protocol under a large variety of conditions, with two exceptions: when messages are multicast to a single group (all protocols perform similarly) and to all destinations (the non-genuine protocol performs better). In LAN environments FastCast performs better than the two other protocols when messages are multicast to few destinations.

The rest of the paper is organized as follows. Section 2 introduces the system model and definitions. Sections 3 and 4 present BaseCast and FastCast, respectively. Section 5 describes our experimental evaluation. Section 6 surveys related work and Section 7 concludes the paper. The appendix contains all proofs of correctness.

2 Model and definitions

In this section, we detail our system model (§2.1) and recall the definitions of consensus (§2.2), and reliable and atomic multicast (§2.3).

2.1 Processes, groups, and links

We consider a system $\Pi = \{p_1, ..., p_n\}$ of processes. Processes communicate by exchanging messages and do not have access to a shared memory or a global clock. The system is asynchronous: messages may experience arbitrarily large (but finite) delays and there is no bound on relative process speeds.

We assume that processes may fail by crashing (i.e., no malicious behavior). A process that never crashes 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 $\bigcup_{g\in\Gamma} g = \Pi$. We assume each group contains $2f+1$ processes, where f is the maximum number of faulty processes per group.

Communication links are fair-lossy, i.e., links do not create, corrupt, or duplicate messages, and guarantee that for any two *correct* processes p and q, and any message m, if p sends m to q infinitely many times, then q receives m an infinite number of times.

2.2 Consensus

We assume the existence of a uniform consensus service in each group g. The consensus service allows processes to propose values and ensures that eventually one of the proposed values is decided. A process in group g proposes a message (or a set of messages) x in instance i by invoking propose$_g[i](x)$, and decides on y in instance i with decide$_g[i](y)$. The consensus service satisfies the following properties:

- *uniform integrity:* if a process decides x in instance i, then x was previously proposed by some process in i.

- *termination:* if a correct process in group g proposes a value in instance i, then every correct process in g eventually decides exactly one value in i.
- *uniform agreement:* if a process in group g decides x in instance i, then no process in g decides $y \neq x$ in i.

To make consensus solvable in each group [12], we further assume that processes at each group have access to a weak leader election oracle [6]. The oracle outputs a single process denoted $leader_{g,p}$ such that there is (a) a correct process l_g in g and (b) a time after which, for every p in g $leader_{g,p} = l_g$.

2.3 Reliable and Atomic Multicast

For every message m, $m.dst$ denotes the groups to which m is multicast. If $|m.dst| = 1$ we say that m is a *local* message; if $|m.dst| > 1$ we say that m is *global*. A process reliably multicasts a message m by invoking primitive r-multicast(m) and delivers m with primitive r-deliver(m).

In this paper, we use non-uniform FIFO reliable multicast, which ensures the following properties:

- *validity*: if a correct process p r-multicasts a message m, then eventually all correct processes $q \in g$, where $g \in m.dst$, r-deliver m.
- *non-uniform agreement*: if a correct process p r-delivers a message m, then eventually all correct processes $q \in g$, where $g \in m.dst$, r-deliver m.
- *integrity*: for any process p and any message m, p r-delivers m at most once, and only if $p \in g$, $g \in m.dst$, and m was previously r-multicast.
- *FIFO order*: if a process r-multicasts a message m before it r-multicasts a message m', then no process r-delivers m' unless it has previously r-delivered m.

With uniform atomic multicast, a process atomically multicasts message m using primitive a-multicast(m) and delivers m with a-deliver(m). We define the relation $<$ on the set of messages processes a-deliver as follows: $m < m'$ iff there exists a process that a-delivers m before m'.

Atomic multicast satisfies the uniform integrity and validity properties of reliable multicast as well as the following properties:

- *uniform agreement*: if a process p a-delivers a message m, then eventually all correct processes $q \in m.dst$ a-deliver m.
- *uniform prefix order*: for any two messages m and m' and any two processes p and q such that $p \in g$, $q \in h$ and $\{g, h\} \subseteq m.dst \cap m'.dst$, if p a-delivers m and q a-delivers m', then either p a-delivers m' before m or q a-delivers m before m'.
- *uniform acyclic order*: the relation $<$ is acyclic.

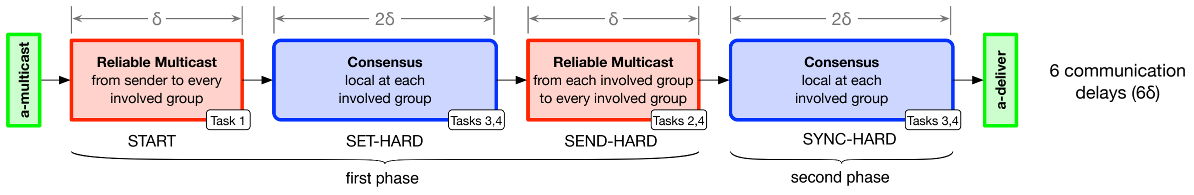
Atomic broadcast is a special case of atomic multicast in which every message is addressed to all groups.

We require reliable and atomic multicast protocols to be *genuine* [14]: an algorithm $\mathscr{A}$ solving reliable or atomic multicast is genuine if and only if for any admissible run R of $\mathscr{A}$ and for any process p in R, if p sends or receives a message, then some message m is r-multicast (resp., a-multicast), and either (a) p is the process that r-multicasts (resp., a-multicasts) m or (b) $p \in g$ and $g \in m.dst$.

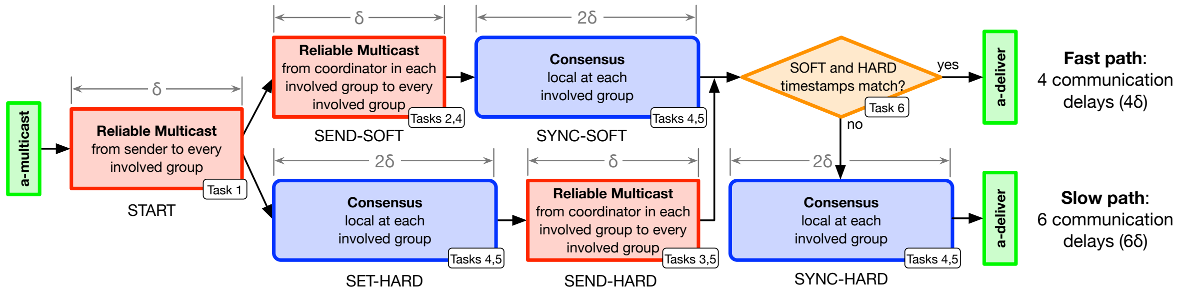
In [14], the authors show the impossibility of solving genuine atomic multicast with weak synchronous assumptions (i.e., unreliable failure detectors [6]) when groups intersect. Hence, we assume that groups are disjoint.

3 Baseline Atomic Multicast

Fast Atomic Multicast, the main contribution of this paper, is inspired by earlier atomic multicast protocols [5, 13, 25]. In this section, we present an early atomic multicast protocol, which we dub *BaseCast*. We initially provide an overview of BaseCast (§3.1), then describe it in detail (§3.2), and reason about its time complexity (§3.3).



(a) BaseCast


a-multicast
δ
Reliable Multicast
from sender to every involved group
Task 1
START
δ
Reliable Multicast
from coordinator in each involved group to every involved group
Tasks 2,4
SEND-SOFT
2δ
Consensus
local at each involved group
Tasks 4,5
SYNC-SOFT
SOFT and HARD timestamps match?
Task 6
yes
no
a-deliver
Fast path:
4 communication delays (4δ)
2δ
Consensus
local at each involved group
Tasks 4,5
SET-HARD
δ
Reliable Multicast
from coordinator in each involved group to every involved group
Tasks 3,5
SEND-HARD
2δ
Consensus
local at each involved group
Tasks 4,5
SYNC-HARD
a-deliver
Slow path:
6 communication delays (6δ)

(b) FastCast

Figure 1: Diagrammatic representations of BaseCast (top) and FastCast (bottom). Existing reliable multicast and consensus algorithms can propagate messages in one communication delay and reach a decision in two communication delays, respectively (e.g., [6] and [19]).

3.1 Overview

In BaseCast, each process implements a logical clock [18] and assigns timestamps to messages based on the logical clock. The correctness of BaseCast stems from two basic properties: (i) processes in the destination of an a-multicast message first assign tentative timestamps to the message and eventually agree on the message's final timestamp; and (ii) processes a-deliver messages according to their final timestamp.

It is easier to understand how BaseCast guarantees properties (i) and (ii) by first considering the special case in which each group g_i in the system has a single and correct process p_i (i.e., $g_i = \{p_i\}$).

(i) To a-multicast a message m to a set of destinations (i.e., groups in $m.dst$), p_i r-multicasts m to the destinations in a START message. Upon r-delivering a START message with m, each destination updates its logical clock, assigns a hard tentative timestamp to m (the reason this timestamp is *hard* is explained in the next section), stores m and its timestamp in a buffer, and r-multicasts m's timestamp to all destinations in a SEND-HARD message. Upon r-delivering timestamps from all destinations in $m.dst$, a process p_i computes m's final timestamp as the maximum among all r-delivered hard tentative timestamps for m.

(ii) Messages are a-delivered respecting the order of their final timestamp. Thus, p_i a-delivers m when it can ascertain that m's final timestamp is smaller than the final timestamp of any messages p_i will a-deliver after m (intuitively, this holds because logical clocks are monotonically increasing).

To handle groups with any number of processes and thereby tolerate process failures, BaseCast uses consensus within each group to ensure that processes in the same group evolve through the same sequence of state changes and produce the same outputs [18, 28]. Consensus is needed within a group to order START messages (we call this consensus a SET-HARD step) and SEND-HARD messages (we call this consensus a SYNC-HARD step). Ordering SET-HARD and SYNC-HARD events within a group ensures that processes in the group assign the same hard tentative timestamp to an a-multicast message m and update their logical clock in the same deterministic way upon handling hard tentative timestamps from m's destination groups.

We refer to the propagation of the START message followed by the SET-HARD step and the propagation of SEND-HARD messages as the *first phase* of the algorithm, and to the SYNC-HARD step as the *second phase* of the algorithm (see Figure 1(a)).

3.2 Detailed algorithm

Algorithm 1 contains five tasks that execute in isolation. The algorithm contains six variables: C_H implements a process's logical clock, used to assign hard tentative timestamps to messages; $\mathscr{B}$ contains timestamps assigned to messages not yet a-delivered; k_p and k_d index consensus instances; and sequences *ToOrder* and *Ordered* are used to totally order messages among processes in a group. Let S and R be two sequences. $S \oplus R$ denotes S followed by R and $S \setminus R$ denotes S without the entries that exist in R.

To a-multicast a message m, process p in group g r-multicasts m to m's destinations using a $(\text{START}, \bot, \bot, m)$ message. When p r-delivers message $(\text{START}, \bot, \bot, m)$ in Task 1 (respectively, $(\text{SEND-HARD}, h, x, m)$ in Task 2), p adds $(\text{SET-HARD}, g, \bot, m)$ (respectively, $(\text{SYNC-HARD}, h, x, m)$) to *ToOrder* to be ordered by consensus in Tasks 3 and 4. Consensus instances are independent and can execute concurrently. However, decision events are handled sequentially according to the order determined by k_d in Task 4.

Process p handles a $(\text{SET-HARD}, \bot, \bot, m)$ tuple in Task 4 by choosing a tentative hard timestamp for m, given by C_H, and propagating the chosen timestamp to m's destinations using a $(\text{SEND-HARD}, g, C_H, m)$ message, if m is global; if m is local, p adds the chosen timestamp to $\mathscr{B}$ as a $(\text{SYNC-HARD}, g, C_H, m)$ tuple. Process p handles a $(\text{SYNC-HARD}, h, x, m)$ tuple by updating its local clock C_H and including the tuple in $\mathscr{B}$.

In Task 5, if p has received tentative timestamps from all groups in m's destinations (i.e., SYNC-HARD tuples in $\mathscr{B}$), p determines m's *final* timestamp as the maximum timestamp among the tentative timestamps assigned to m by m's destinations. Process p a-delivers m when it ascertains that no a-multicast message m' will have a final timestamp smaller than m's. This happens when no a-multicast message m' r-delivered by p has (a) a final timestamp smaller than m's and (b) a tentative timestamp smaller than m's final timestamp, if p has not received tentative timestamps from all of m''s destinations yet.

3.3 Time complexity

We now state the best-case time complexity between a-multicast(m) and a-deliver(m) in Algorithm 1 in terms of δ, the maximum communication delay.

Proposition 1 *Let Algorithm 1 use the reliable multicast implementation in [6] and the consensus implementation in [19]. Every atomically multicast global message is delivered in at least* 6δ.

4 Fast Atomic Multicast

In this section, we introduce *FastCast*, a genuine atomic multicast algorithm that can deliver global messages *fast*, in four communication delays. We first provide a general description of FastCast (§4.1), then present it in detail (§4.2) and discuss its time complexity (§4.3).

4.1 Overview

Fast Atomic Multicast implements two execution paths for global messages, a *fast path* and a *slow path*. The two paths execute concurrently and are triggered after a message m is a-multicast (see Figure 1(b)).

In the fast path, processes in each group $g \in m.dst$ assign a soft tentative timestamp to m, which is a "guess" of what g's hard tentative timestamp will be. We call this a guess because g's hard tentative timestamp is determined after processes in g execute consensus, in a SET-HARD step. Processes r-multicast m's soft tentative timestamp to destinations in $m.dst$ in a SEND-SOFT message. Upon r-delivering soft tentative timestamps, processes in g execute consensus to ensure that they act in the same way. We call this consensus a SYNC-SOFT step.

Processes in g can a-deliver m fast if two conditions are satisfied: (a) processes have r-delivered and ordered (through consensus) soft tentative timestamps from all groups in $m.dst$, and therefore they can compute m's final timestamp; and (b) the soft tentative timestamp of each $g \in m.dst$ matches g's hard tentative timestamp (determined in the slow path).

The slow path resembles BaseCast, with the exception that processes may skip the second phase of the protocol. When processes in g have collected soft tentative timestamps, as a result of the fast path, and hard tentative timestamps, as a result of the first phase of the slow path, from all groups in $m.dst$, they compare the timestamps. If the soft and hard tentative timestamps for each group in $m.dst$ match, then processes can a-deliver m without executing the second phase of the slow path. If the timestamps do not

```
**Algorithm 1: Baseline Atomic Multicast - BaseCast** (for process $p$ in group $g$)

Initialization
    $C_H \leftarrow 0$                                                   {$p$'s logical clock}
    $\mathscr{B} \leftarrow \emptyset$                                   {contains tentative timestamps of undelivered messages}
    $k_p \leftarrow 0$; $k_d \leftarrow 0$                               {contain the consensus instance for propose and decide events, respectively}
    $ToOrder \leftarrow \epsilon$; $Ordered \leftarrow \epsilon$         {sequences of tuples to be ordered and already ordered, respectively}

To a-multicast message $m$:
    r-multicast $(\text{START}, \perp, \perp, m)$ to $m.dst$

**when** r-deliver$(\text{START}, \perp, \perp, m)$                     {**Task 1**}
    $ToOrder \leftarrow ToOrder \oplus (\text{SET-HARD}, g, \perp, m)$   {add a tuple to be ordered by consensus}

**when** r-deliver$(\text{SEND-HARD}, h, x, m)$                         {**Task 2**}
    **if** $\forall y : (\text{SYNC-HARD}, h, y, m) \notin ToOrder$ **then**   {if this tuple hasn't been included already...}
        $ToOrder \leftarrow ToOrder \oplus (\text{SYNC-HARD}, h, x, m)$  {...add it to be ordered}

**when** $ToOrder \setminus Ordered \neq \emptyset$                     {**Task 3**}
    propose$_g[k_p](ToOrder \setminus Ordered)$                          {propose all tuples that haven't been ordered yet}
    $k_p \leftarrow k_p + 1$                                             {adjust counter for next propose instance}

**when** decide$_g[k_d](Decided)$                                       {**Task 4**}
    **for** each $z, h, x, m : (z, h, x, m) \in Decided \setminus Ordered$ in order **do**   {for each ordered tuple}
        **if** $z = \text{SET-HARD}$ **then**                            {set $g$'s hard tentative timestamp}
            $C_H \leftarrow C_H + 1$                                     {increment logical clock}
            **if** $|m.dst| > 1$ **then**                                {if $m$ is a global message:}
                $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\text{SEND-HARD}, g, C_H, m)\}$   {store $g$'s hard tentative timestamp assigned to $m$ and...}
                r-multicast $(\text{SEND-HARD}, g, C_H, m)$ to $m.dst$   {...send it to all of $m$'s destinations}
            **else**                                                     {if $m$ is a local message:}
                $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\text{SYNC-HARD}, g, C_H, m)\}$   {$g$'s hard tentative timestamp is ordered}
        **if** $z = \text{SYNC-HARD}$ **then**                           {handle the receipt of $h$'s hard tentative timestamp}
            $C_H \leftarrow max(\{C_H, x\})$                             {Lamport's rule to update logical clocks}
            $\mathscr{B} \leftarrow \mathscr{B} \setminus \{(\text{SEND-HARD}, h, x, m)\}$   {no longer needed since will store ordered timestamp}
            $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\text{SYNC-HARD}, h, x, m)\}$        {$h$'s hard tentative timestamp is now ordered}
        $Ordered \leftarrow Ordered \oplus (z, h, x, m)$                 {this tuple has been handled}
    $k_d \leftarrow k_d + 1$                                             {adjust counter for next decide instance}

**when** $\exists m \; \forall h \in m.dst \; \exists x : (\text{SYNC-HARD}, h, x, m) \in \mathscr{B}$   {**Task 5**}
    $ts \leftarrow max(\{x : (\text{SYNC-HARD}, h, x, m) \in \mathscr{B}\})$   {$ts$ is $m$'s final timestamp}
    **for** each $z, h, x : (z, h, x, m) \in \mathscr{B}$ **do** $\mathscr{B} \leftarrow \mathscr{B} \setminus \{(z, h, x, m)\}$   {replace $m$'s tentative timestamps in $\mathscr{B}$ by...}
    $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\text{FINAL}, \perp, ts, m)\}$   {...$m$'s final timestamp}
    **while** $\exists(\text{FINAL}, \perp, ts, m) \in \mathscr{B} : \forall(z, h, x, m') \in \mathscr{B}, m \neq m' : ts < x$ **do**   {for each deliverable message}
        a-deliver $m$                                                    {deliver it!}
        $\mathscr{B} \leftarrow \mathscr{B} \setminus \{(\text{FINAL}, \perp, ts, m)\}$   {$\mathscr{B}$ only contains undelivered messages}
```

match for some group, processes continue the execution of the second phase of the slow path, just like in BaseCast.

We now modify the procedure above so that processes in a group can make "educated guesses" of their hard tentative timestamps, and thereby a-deliver messages fast.

- In each group g, consensus is implemented with Paxos [19]. In each Paxos instance i, the leader in g proposes the value to be decided in instance i. In runs in which there is a single and correct leader in g, processes will decide on the value proposed by the leader [19].
- The leader is the only process in a group to propagate soft tentative timestamps (SEND-SOFT messages) to the destination groups of a message and uses the chosen timestamp as the proposed hard tentative timestamp in consensus (SET-HARD step).

As a result, in executions in which there is a single and correct leader in each group, a-multicast messages will be a-delivered fast.

4.2 Detailed algorithm

Algorithm 2 contains seven tasks that execute in isolation. In addition to the variables used in Algorithm 1, it also implements a logical clock C_S used to assign soft tentative timestamps to messages. In the following, we first describe the execution of global messages and then the execution of local messages.

A process a-multicasts global message m by r-multicasting (START, $\perp$, $\perp$, m) to m's destinations (i.e., groups in $m.dst$). In Task 1, process p in group g r-delivers the START message and requests processes in g to compute a hard tentative timestamp for m, by adding (SET-HARD, g, $\perp$, m) to sequence *ToOrder*. Entries in *ToOrder* are ordered by consensus in Tasks 4 and 5. In Task 4, if p is the leader of g, it proposes tuples in *ToOrder* in the next available consensus instance; p also chooses a soft tentative timestamp for m and r-multicasts this timestamp in tuple (SEND-SOFT, h, C_S, m) to m's destinations.

Soft tentative timestamps are r-delivered in Task 2, where p adds (SYNC-SOFT, h, x, m) to *ToOrder* to be ordered by consensus in Task 4. Task 5 is responsible for consensus decisions, handling them sequentially and in order. To handle a (SET-HARD, h, x, m) tuple, p updates its logical clock C_H, picks a hard tentative timestamp for m and r-multicasts this timestamp to m's destinations in a SEND-HARD message. Finally, p handles (SYNC-HARD, h, x, m) and (SYNC-SOFT, h, x, m) tuples by updating its logical clock C_H and adding the tuples to $\mathscr{B}$.

A tuple (SEND-HARD, h, x, m), with m's hard tentative timestamp from group h, is r-delivered in Task 3. As a result, p adds (SYNC-HARD, h, x, m) to *ToOrder* so that it is ordered by consensus, as described above. Task 6 checks whether the soft and hard tentative timestamps for m match, in which case it sets m as ordered and places m in $\mathscr{B}$, where it will be eventually a-delivered. The condition for a-delivering m is similar in Algorithm 1 (Task 5) and Algorithm 2 (Task 7).

Local a-multicast messages experience a shorter execution path than global messages. When a message m is a-multicast to a single group g, it is r-multicast to the members of g in a (START, $\perp$, $\perp$, m) message. Upon r-delivering m, g's members execute a round of consensus (SET-HARD step) to agree on the hard tentative timestamp to assign to m and ensure that future messages will be assigned a timestamp greater than the one assigned to m. When the proposed timestamp is decided, it becomes m's final timestamp.

4.3 Time complexity

Proposition 2 states that Algorithm 2 can deliver messages fast, in four communication delays.

Proposition 2 *Assume Algorithm 2 uses the reliable multicast implementation in [6] and the consensus implementation in [19]. If $\mathscr{R}$ is a set of runs of Algorithm 2 in which for every group g there is a correct leader l_g and for each $p \in g$ $leader_{g,p} = l_g$, then there are runs in $\mathscr{R}$ in which atomically multicast global messages are delivered fast, in four communication delays.*

5 Performance evaluation

In this section, we describe the main motivations that guided the design of our experiments (§5.1), detail the environments in which we conducted the experiments (§5.2), explain how we implemented a social network benchmark (§5.3), and then present and discuss the results (§5.4–5.7). We conclude with a summary of the main findings (§5.8).

```
**Algorithm 2: Fast Atomic Multicast - FastCast** (for process $p$ in group $g$)

Initialization
  $C_H \leftarrow 0$                                          {p's logical clock, used for hard tentative timestamps}
  $C_S \leftarrow 0$                                          {p's logical clock, used for soft tentative timestamps}
  $\mathscr{B} \leftarrow \emptyset$                          {contains tentative timestamps of undelivered messages}
  $k_p \leftarrow 0$; $k_d \leftarrow 0$                      {contain the consensus instance for propose and decide events, respectively}
  $ToOrder \leftarrow \epsilon$; $Ordered \leftarrow \epsilon$  {sequences of tuples to be ordered and already ordered, respectively}

To a-multicast message $m$:
  r-multicast $(\textsc{start}, \perp, \perp, m)$ to $m.dst$

**when** r-deliver$(\textsc{start}, \perp, \perp, m)$                              {**Task 1**}
  $ToOrder \leftarrow ToOrder \oplus (\textsc{set-hard}, g, \perp, m)$            {add a tuple to be ordered by consensus}

**when** r-deliver$(\textsc{send-soft}, h, x, m)$                                  {**Task 2**}
  **if** $\forall y : (\textsc{sync-soft}, h, y, m) \notin ToOrder$ **then**      {if this tuple hasn't been included already...}
    $ToOrder \leftarrow ToOrder \oplus (\textsc{sync-soft}, h, x, m)$            {...add it to be ordered}

**when** r-deliver$(\textsc{send-hard}, h, x, m)$                                  {**Task 3**}
  **if** $\forall y : (\textsc{sync-hard}, h, y, m) \notin ToOrder$ **then**      {if this tuple hasn't been included already...}
    $ToOrder \leftarrow ToOrder \oplus (\textsc{sync-hard}, h, x, m)$            {...add it to be ordered}

**when** $ToOrder \setminus Ordered \neq \emptyset$ **and** $leader_{g,p} = p$      {**Task 4**}
  $C_S \leftarrow \max(\{C_H, C_S\})$                            {soft timestamp should not be smaller than hard timestamp}
  **for** each $(z, h, x, m) \in ToOrder \setminus Ordered$ in order **do**     {for each unordered message}
    **if** z = $\textsc{set-hard}$ **then**
      $C_S \leftarrow C_S + 1$                                    {increment logical clock}
      **if** $|m.dst| > 1$ **then** r-multicast $(\textsc{send-soft}, h, C_S, m)$ to $m.dst$   {propagate soft tentative timestamp}
    **else**
      $C_S \leftarrow \max(\{C_S, x\})$                          {soft timestamp should not be smaller than unordered timestamps}
  propose$_g[k_p](ToOrder \setminus Ordered)$                     {propose all tuples that haven't been ordered yet}
  $k_p \leftarrow k_p + 1$                                        {adjust counter for next propose instance}

**when** decide$_g[k_d](Decided)$                                                {**Task 5**}
  **for** each $(z, h, x, m) \in Decided \setminus Ordered$ in order **do**     {for each ordered tuple}
    **if** $z = \textsc{set-hard}$ **then**                                     {set g's hard tentative timestamp}
      $C_H \leftarrow C_H + 1$                                    {increment logical clock}
      **if** $|m.dst| > 1$ **then** r-multicast $(\textsc{send-hard}, g, C_H, m)$ to $m.dst$   {send hard tentative timestamp}
      **else** $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\textsc{sync-hard}, g, C_H, m)\}$    {if m is local, hard tentative timestamp is ordered}
    **if** $z = \textsc{sync-soft}$ **then**
      $C_H \leftarrow \max(\{C_H, x\})$                          {Lamport's rule to update logical clocks}
      $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\textsc{sync-soft}, h, x, m)\}$   {h's soft tentative timestamp is now ordered}
    **if** $z = \textsc{sync-hard}$ **then**
      $C_H \leftarrow \max(\{C_H, x\})$                          {Lamport's rule to update logical clocks}
      $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\textsc{sync-hard}, h, x, m)\}$   {h's hard tentative timestamp is now ordered}
    $Ordered \leftarrow Ordered \oplus \{(z, h, x, m)\}$          {this tuple has been handled}
  $k_d \leftarrow k_d + 1$                                        {adjust counter for next decide instance}

**when** $\exists h, x, m : (\textsc{sync-soft}, h, x, m) \in \mathscr{B}$ **and**
                 $(\textsc{sync-hard}, h, x, m) \in ToOrder \setminus Ordered$     {**Task 6**}
  $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\textsc{sync-hard}, h, x, m)\}$     {h's hard tentative timestamp is now ordered}
  $Ordered \leftarrow Ordered \oplus \{(\textsc{sync-hard}, h, x, m)\}$          {this tuple has been handled}

**when** $\exists m\ \forall h \in m.dst\ \exists x : (\textsc{sync-hard}, h, x, m) \in \mathscr{B}$   {**Task 7**}
  $ts \leftarrow max(\{x : (\textsc{sync-hard}, h, x, m) \in \mathscr{B}\})$     {$ts$ is m's final timestamp}
  **for** each $z, h, x : (z, h, x, m) \in \mathscr{B}$ **do** $\mathscr{B} \leftarrow \mathscr{B} \setminus \{(z, h, x, m)\}$   {replace m's tentative timestamps in $\mathscr{B}$ by...}
  $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\textsc{final}, \perp, ts, m)\}$   {...m's final timestamp}
  **while** $\exists(\textsc{final}, \perp, ts, m) \in \mathscr{B}\ \forall(z, h, x, m') \in \mathscr{B}, m \neq m' : ts < x$ **do**   {for each deliverable message}
    a-deliver $m$                                                 {deliver it!}
    $\mathscr{B} \leftarrow \mathscr{B} \setminus \{(\textsc{final}, \perp, ts, m)\}$   {$\mathscr{B}$ only contains undelivered messages}
```

5.1 Evaluation rationale

In the following, we explain our choices for environments, benchmarks, and protocols.
Environments. We consider a LAN, an emulated WAN, and a real WAN. The LAN and emulated WAN provide controlled environments, where experiments can run in isolation; the real WAN represents a setting in which we expect FastCast to be used in practice. FastCast optimizes for the number of communication delays. Thus, we conjecture that the protocol performs well in environments in which latencies are high. We test our conjecture in the emulated and real WANs. We would also like to know how FastCast performs in less favorable environments, where the difference between communication and processing delays is less significant (i.e., LAN).
Benchmarks. We use a microbenchmark with 64-byte messages to evaluate particular scenarios in isolation. We vary the number of groups (up to 16 groups, the largest configuration we can accommodate in our local infrastructure) and the number of message destinations. We use a social network service to define message destinations according to user connections in a social network graph. In these two benchmarks, we consider executions with a single client to understand the performance of FastCast without queueing effects and executions with multiple clients to evaluate FastCast under stress.
Protocols. We compare FastCast to another genuine atomic multicast protocol, BaseCast, and to a non-genuine atomic multicast protocol that uses a fixed group of processes to order messages, regardless of the message destinations. The fixed group of processes uses MultiPaxos to order atomically multicast messages. Although the non-genuine protocol does not scale, under low load (i.e., with a single client) it provides a performance reference since messages can be ordered in three communication delays.

5.2 Implementation and environments

We implemented prototypes of BaseCast, FastCast and the non-genuine atomic multicast in C. For brevity, hereafter we refer to the non-genuine atomic multicast protocol as MultiPaxos. We used libpaxos,[1] a Multi-Paxos C library, for consensus, and implemented point-to-point communication with TCP. A stable leader for each group is defined prior to the execution, which is expected to be the common case.

Local-area network (LAN). This environment consisted of a set of nodes, each node with an eight-core Intel Xeon L5420 processor working at 2.5GHz, 8GB of memory, SATA SSD disks, and 1Gbps ethernet card. Each node runs CentOS 7.1 64 bits. The RTT (round-trip time) between nodes in the cluster is around 0.1ms.

Emulated wide-area network (emulated WAN). For these experiments, we use the LAN environment and divide nodes in three "regions", R1, R2 and R3. The latencies between nodes in different regions were emulated using Linux traffic control tools. We used latency values measured in a real WAN (see below), with average RTT of 70ms (R1⟷R2), 70ms (R2⟷R3), and 144ms (R1⟷R3), and standard deviation of 5%.

Real wide-area network (WAN). We used Amazon EC2, a public wide-area network. All the nodes are m3.large instances, with 2 vCPUs and 7.5GB of memory. We allocated nodes in three regions: California (R1), North Virginia (R2) and Ireland (R3).

In all experiments, groups contain three processes, each process running in a different node. In the WAN setups, we distribute clients in three regions and deploy each process in a group in a different region (see Figure 2). Consequently, each group can tolerate the failure of a whole datacenter.

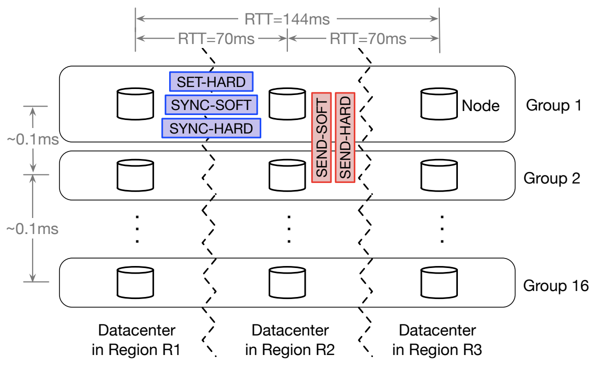


Figure 2: Configuration in WAN.

[1] http://libpaxos.sourceforge.net/paxos_projects.php#libpaxos3

5.3 Social network benchmark

We developed a social network service similar to Twitter. Users can follow and unfollow other users, post a message, and read the last messages posted by the users they follow. We created a social network graph with ten thousand users and then partitioned the graph in 16 partitions using METIS,[2] a popular graph partitioner. METIS strives to balance the number of users per partition while minimizing the number of edges across partitions. In our social graph, 7110 users have followers in the same partition, 2474 users have followers in two partitions, 376 users have followers in three partitions, and the remaining 40 users have followers in four or five partitions. When a user posts a message (64 bytes), the message is atomically multicast to all the groups that contain followers of the user. This ensures that reads are single-group operations. Since reads and posts are implemented with atomic multicast, our social network service offers strong consistency guarantees. In the experiments, we run posts only, since as mentioned above, reads are always local to a partition.

5.4 Microbenchmark in LAN

The first experiment explores the advantage of genuine atomic multicast algorithms over non-genuine approaches like MultiPaxos. Figure 3 shows the throughput in messages per second versus the number of groups, when 200 clients per group multicast local messages only. The results show that genuine atomic multicast protocols result in throughput that increases linearly with the number of groups, from 36000 messages per second with 1 group to almost 600000 messages per second with 16 groups. This happens because genuine protocols only involve the sender of a message and the destination group. Because MultiPaxos has to order all the messages and is nearly saturated with 200 client, performance with 2 groups is only slightly higher than with 1 group, reaching a maximum of 48000 messages per second. BaseCast and FastCast have identical results because the mechanism to order messages addressed to one group is similar in both algorithms. Even though MultiPaxos does not scale, it provides a useful reference for latency with few clients (i.e., low load), since it orders messages in three communication delays.

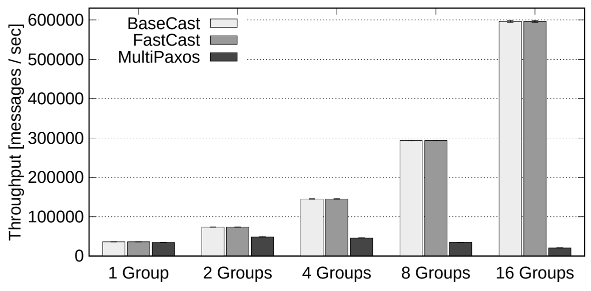


Figure 3: Throughput for single-group messages in a LAN.

The next experiments assess latency with a single client. This setup aims to check if the reduction in communication delays introduced by FastCast has the expected impact on latency. We consider configurations with an increasing number of groups and clients multicast messages to all groups in the configuration. FastCast's advantage is more noticeable when the number of groups is smaller than 8, with a reduction from 0.928 ms to 0.691ms with 2 groups when compared to BaseCast, and from 1.068 ms to 0.847ms with 4 groups (Figure 4, top left). With 16 groups, the overhead introduced by the slow and fast paths in FastCast impacts latency negatively. MultiPaxos has lower latency in almost all the cases, an expected result that reflects the advantage of atomic broadcast over atomic multicast when the number of destinations is equal or close to the total number of groups [27]. Figure 4 (top right) shows similar results for a fixed number of groups with increasing destinations. Multipaxos has higher latency in this configuration since all processes must be reached, even if the destinations are a subset of all groups.

We now assess the protocols under "operational load", that is, with enough clients to stress the system, without saturating resources. In these experiments, the number of groups is fixed to 16 and we vary the number of destinations in multicast messages. We keep the product *number of destination groups* × *number of clients* equal to 1536. The rationale is that the cost to multicast a message grows with the number of

[2]http://glaros.dtc.umn.edu/gkhome/views/metis

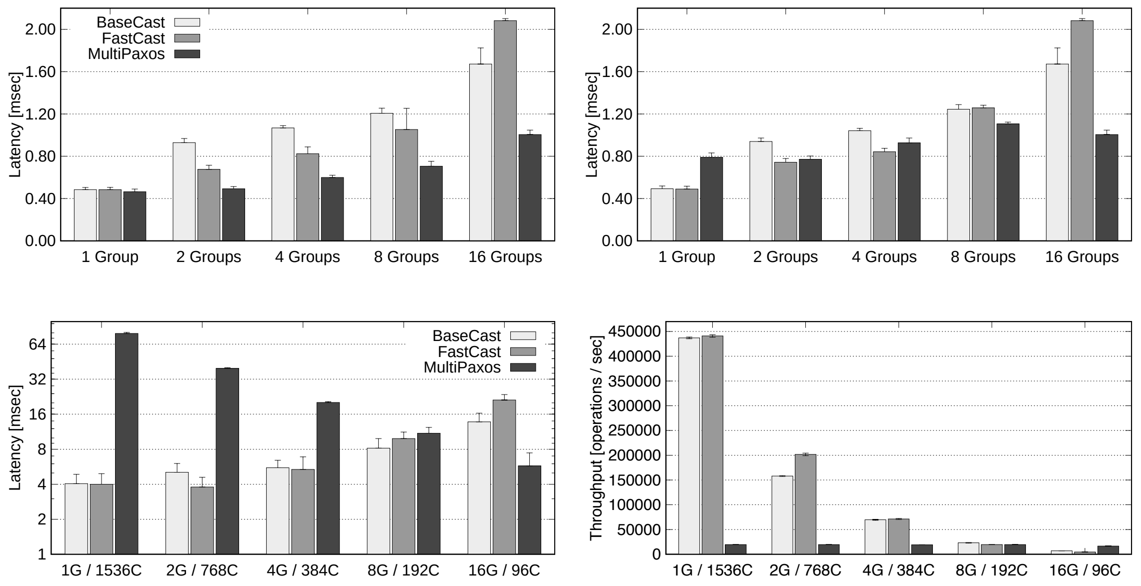


Figure 4: Atomic multicast in a LAN. Bars show median latency or average throughput, and whiskers show 95-th percentile or 95% confidence interval. (Top left.) Latency when one client multicasts to all groups versus number of groups in configuration. (Top right.) Latency when one client multicasts to k groups in a system with 16 groups. (Bottom left and right.) Latency and throughput when multiple clients multicast to increasing number of destination groups in a system with 16 groups. Legend in x-axis shows number of destination groups k_g in a multicast message and number of clients k_c in the configuration, where $k_g \times k_c = 1536$.

destinations. Thus, to avoid overloading the system, we decrease the number of clients as we increase the number of destination groups.

Figure 4 shows latency and throughput (graphs at the bottom). FastCast outperforms BaseCast for 2 destination groups, with a latency and throughput of 3.8 ms and 202000 messages per second against 5.1 ms and 158000 messages per second for BaseCast. With more destinations, the overhead introduced by FastCast's parallel paths execution make it less efficient than BaseCast, which has a single execution path. MultiPaxos is CPU-bound with more than 200 clients.

5.5 Microbenchmark in emulated WAN

We now test our conjecture that FastCast is more suitable for WANs. Initially, we assess the latency of the three protocols in the absence of queuing effects. A single client atomically multicasts, in closed loop, global messages. The expected latency for the execution of a Paxos instance in libpaxos is around 70 ms, which is the round-trip time (RTT) between the two closest regions. This is time enough for the Paxos coordinator to receive responses from a quorum of acceptors.

Figure 5 (top left) shows that for any number of destination groups, MultiPaxos and FastCast have a median latency around 1 RTT, while BaseCast has always 2 RTT latency. Although FastCast executes two consensus instances for each global message, just like BaseCast, one consensus is executed in the fast path and the other consensus in the slow path. Since both paths are executed in parallel, FastCast has similar latency as MultiPaxos. The reduced communication steps, in such scenario, makes FastCast twice as fast as BaseCast. Differently from the results found in a LAN, FastCast largely outperforms BaseCast in configurations with 16 groups when increasing the number of destinations (Figure 5, top right).

We now consider experiments with multiple clients and 16 groups (Figure 5, bottom graphs). As in the LAN experiments, we keep the *number of destinations* × *number of clients* factor constant for each configuration. In these experiments, we also evaluate the performance of FastCast when messages are ordered through the slow path. For this case, we changed the protocol to force the leader to make wrong timestamp guesses.

FastCast consistently outperforms BaseCast for global messages up to 8 destination groups. With 2 destination groups, the throughput of FastCast is 70% higher than BaseCast's (8400 versus 5050 messages per

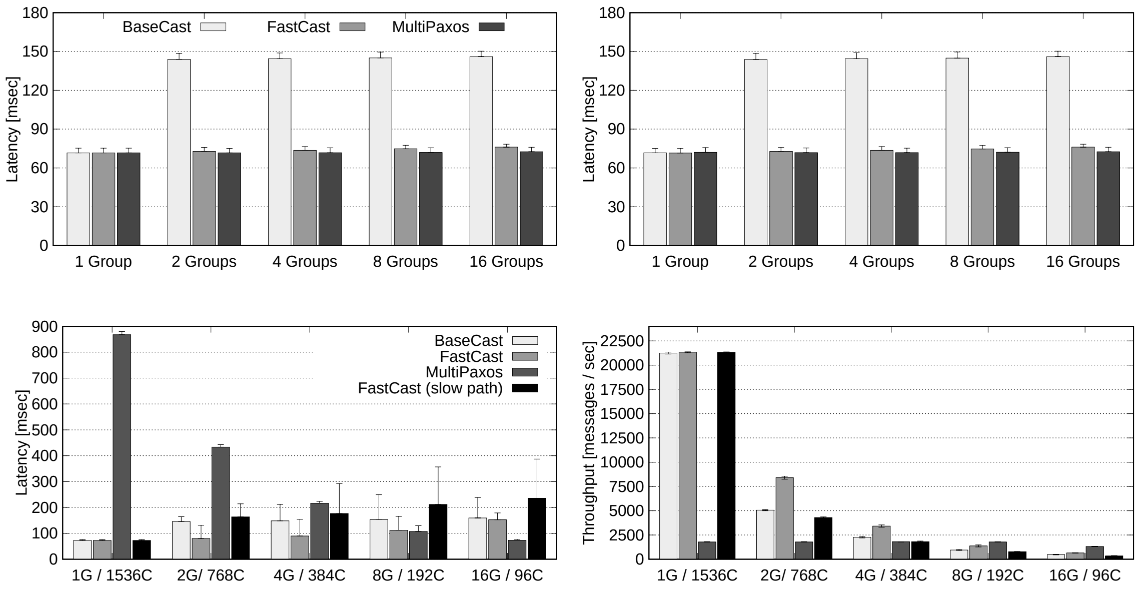


Figure 5: Atomic multicast in emulated WAN. Bars show median latency or average throughput, and whiskers show 95-th percentile or 95% confidence interval. (Top left.) Latency when one client multicasts to all groups versus number of groups in configuration. (Top right.) Latency when one client multicasts to k groups in a system with 16 groups. (Bottom left and right.) Latency and throughput when multiple clients multicast to increasing number of destination groups in a system with 16 groups. Legend in x-axis shows number of destination groups k_g in a multicast message and number of clients k_c in the configuration, where $k_g \times k_c = 1536$.

second on average). When messages are multicast to all 16 groups, FastCast and BaseCast display similar latency, although FastCast has higher throughput than BaseCast, 647 versus 475 messages per second. Both genuine atomic multicast protocols perform much better than the non-genuine protocol up to 8 destination groups, although the advantage decreases as the number of destinations increases. With 8 destination groups, FastCast and MultiPaxos have similar latency, and BaseCast performs worse than the two other protocols. With 16 destination groups, MultiPaxos has lower latency and higher throughput than the other protocols. This fact confirms findings in other works in the literature, which show that when messages address all groups, atomic broadcast protocols have superior performance than atomic multicast protocols [27].

Up to 4 destination groups in a message, FastCast can order messages through the slow path with latency slightly higher than BaseCast's. With 8 and 16 destination groups, the additional overhead of executing three consensus instances in the fast and slow paths degrades the performance of FastCast.

These results establish that, as we anticipated, FastCast shines in environments where communication latency is high. In these environments, FastCast can take full advantage of its reduced number of communication delays. In the next section, we aim to confirm these observations in a real WAN.

5.6 Microbenchmark in real WAN

To confirm the results obtained with the emulated WAN, we executed the same set of experiments on Amazon EC2. We used up to 48 nodes for the maximum of 16 groups and one additional nodes per group to run the clients.

The results for executions with a single client (Figure 6, graphs on the top) are very similar to those we found with the emulated WAN. (We recall that in the emulated WAN we used RTT values between regions that correspond to measurments in the real WAN.) For the experiments with increased load, FastCast had slightly better performance in the real WAN than in the emulated WAN for configurations with 8 and 16 groups. We attribute this improvement to the fact that m3.large instances have better processors than the ones in our LAN and emulated WAN, which reduces FastCast's CPU overhead.

FastCast beats BaseCast in all configuration, with latency of 84 ms for 2, 4 and 8 destination groups and 101 ms for 16 groups. BaseCast has a latency between 163 ms and 170 ms in all cases (Figure 6, bottom left). Regarding throughput, Figure 6 (bottom right) shows that FastCast is 80% faster than BaseCast with

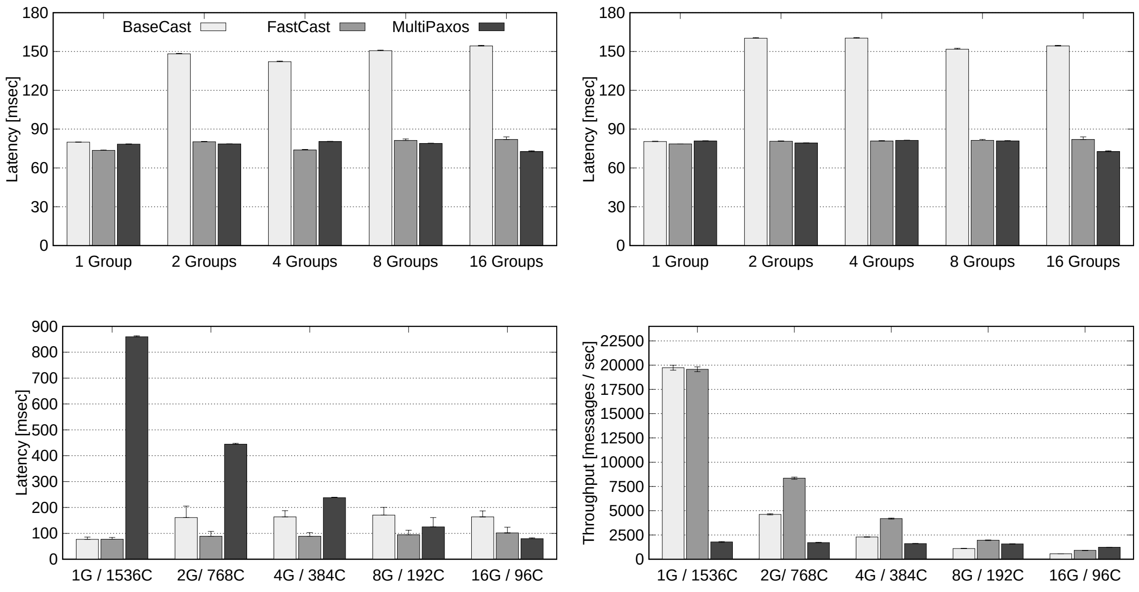


Figure 6: Atomic multicast in a real WAN. Bars show median latency or average throughput, and whiskers show 90-th percentile or 95% confidence interval. (Top left.) Latency when one client multicasts to all groups versus number of groups in configuration. (Top right.) Latency when one client multicasts to k groups in a system with 16 groups. (Bottom left and right.) Latency and throughput when multiple clients multicast to increasing number of destination groups in a system with 16 groups. Legend in x-axis shows number of destination groups k_g in a multicast message and number of clients k_c in the configuration, where $k_g \times k_c = 1536$.

2 destination groups (4600 vs. 8350 messages per second) and more than 60% faster when messages are addressed to all the available groups (565 vs. 913 messages per second). MultiPaxos outperforms the other two protocols when all the groups are in the destination of the messages, delivering 1213 messages per second against the 913 messages per second of FastCast.

5.7 Social network in emulated WAN

In the first experiment, a single client posts messages in a closed loop. Each post operation results in the posted message atomically multicast to all the groups in which the poster has at least one follower. In Figure 7 (top left) we can see that FastCast performs similarly to MultiPaxos, despite the additional consensus instance needed by FastCast to order global messages. FastCast's latency is in the range 73–76ms, which approximates 1 RTT). BaseCast, on the contrary, has latency 2 times greater because of the two sequential consensus executions needed to order global messages.

We now consider executions with an increasing number of clients. Figure 7 (top right) compares the throughput of the three protocols. Up to 3200 clients, that is, 200 clients per group, FastCast outperforms both BaseCast and MultiPaxos. After 3200 clients, FastCast saturates with a throughput of 12500 posts per second. BaseCast eventually reaching FastCast with similar throughput before both systems reach saturation, with 4000 clients.

Figure 7 (both graphs at the bottom) shows the latencies for executions with 800 and 1600 clients. MultiPaxos is overwhelmed in both cases. With 800 clients, the latency of FastCast is close to 1 RTT, between 80ms and 90ms, while the latency of BaseCast is around 150ms. With 1600 clients (100 clients per group), FastCast's latency increases to 130ms, while the latency of BaseCast reaches 200ms.

5.8 Summary

We now summarize the results of our performance evaluation. Table 1 compares the protocols based on the results presented in the previous sections.

We draw the following conclusions from our evaluation.

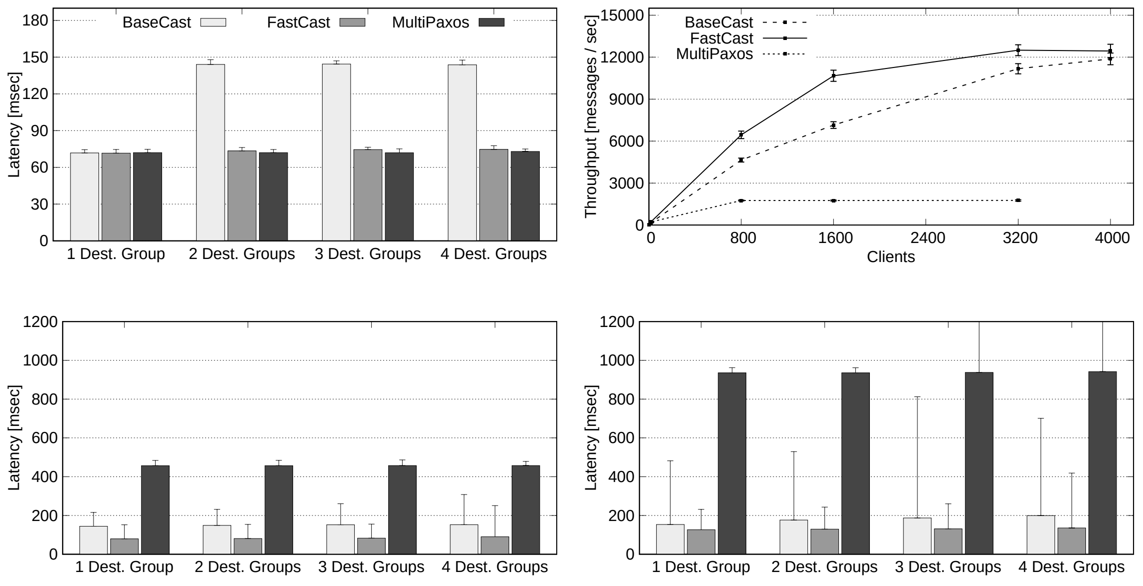


Figure 7: Social network application in an emulated WAN. Bars show median latency, lines show average throughput, and whiskers show 90-th percentile or 95% confidence interval. (Top left.) Latency when one client executes 'post' commands versus number of groups in the client's followers list. (Top right.) Throughput versus number of clients executing 'post' commands. (Bottom left.) Latency when 800 clients execute 'post' commands versus number of groups in the clients' followers list. (Bottom right.) Latency when 1600 clients execute 'post' commands versus number of groups in the clients' followers list.

- FastCast outperforms other protocols in most configurations in a WAN (emulated and real), with a few exceptions: (a) For local messages, under low load all protocols perform the same and under high load FastCast and BaseCast have similar performance, better than MultiPaxos's. (b) For messages addressed to all destinations, MultiPaxos performs best.
- In LAN under high load, no protocol stands out. FastCast performs best with few destination groups, BaseCast with many destination groups, but not all, when MultiPaxos is better than the other two.
- MultiPaxos outperforms the genuine protocols in the LAN with one client, and in all cases when messages are multicast to all destinations, with one exception (microbenchmark in emulated WAN, single client and 16 destinations).

6 Related work

Several multicast and broadcast algorithms have been proposed in the literature [10]. Moreover, many systems ensure strong consistency with "ad hoc" ordering protocolos that do not implement all the properties of atomic multicast (e.g., [8, 29, 17]). We focus next on atomic multicast algorithms.

Existing atomic multicast algorithms fall into one of three categories: *timestamp-based*, *round-based*, and *ring-based*. Algorithms based on timestamps (i.e., [13, 24, 26] and the algorithms proposed in this paper) are all genuine and can be considered variations of an early atomic multicast algorithm [5], designed for failure-free systems. In these algorithms, processes assign timestamps to messages, ensure that destinations agree on the final timestamp assigned to each message, and deliver messages following this timestamp order. The precise way in which these properties are ensured varies from one algorithm to another. The algorithms in [13, 26] have a best-case time complexity of 6δ for the delivery of global messages. The algorithm in [24] can deliver global messages in 5δ and it ensures another property besides genuineness called *message-minimality*. This property states that the messages of the algorithm have a size proportional to the number of destination groups of the multicast message, and not to the total number of processes. The two algorithms proposed in this paper verify this property as well.

Benchmark	Environment	Load	Destinations in system with 16 groups				
			1	2	4	8	16
Micro bechmark	LAN	low	all equal	MP	MP	MP	MP
		high	BC **FC**	**FC**	BC **FC**	BC	MP
	emulated WAN	low	all equal	**FC** MP	**FC** MP	**FC** MP	MP
		high	BC **FC**	**FC**	**FC**	**FC**	MP
	real WAN	low	all equal	**FC** MP	**FC** MP	**FC** MP	MP
		high	BC **FC**	**FC**	**FC**	**FC**	MP
Social Network	emulated WAN	low	all equal	**FC** MP	**FC** MP	Not applicable	
		high	**FC**	**FC**	**FC**	Not applicable	

Table 1: How atomic multicast protocols compare. FC=FastCast, BC=BaseCast, MP=MultiPaxos; each cell shows the best performing protocol(s) in the given configuration.

In round-based algorithms, processes execute an unbounded sequence of rounds and agree on messages delivered at the end of each round. A round-based atomic multicast algorithm that can deliver messages in 4δ is presented in [26]. Differently from FastCast, which can also deliver global messages in 4δ, the algorithm in [26] is not genuine.

Ring-based algorithms propagate messages along a predefined ring overlay and ensure atomic multicast properties by relying on this topology. An atomic multicast algorithm in this category is proposed in [11], where consensus is run among the members of each group. The time complexity of this algorithm is proportional to the number of destination groups.

Multi-Ring Paxos [21], Spread [1, 2], and Ridge [4] are ring-based non-genuine atomic multicast protocols. On the one hand, to deliver a message m, they require communication with processes outside of the destination groups of m. On the other hand, these protocols do not require disjoint groups.

Although FastCast is the first optimistic atomic multicast protocol, optimistic execution to improve performance has been explored before in other contexts, such as atomic broadcast (e.g., [23, 31, 22]) and quorum systems [15].

7 Conclusion

Atomic multicast is a fundamental communication abstraction in the design of scalable and highly available strongly consistent distributed systems. This paper proposes FastCast, the first genuine atomic multicast algorithm that can order local messages, addressed to a single group, in three communication delays and global messages, addressed to multiple groups of processes, in four communication delays. In addition to introducing a novel genuine atomic multicast algorithm, we also assessed its performance in three different environments. The results show that FastCast largely outperforms other genuine and non-genuine atomic multicast protocols.

Acknowledgements

This work is supported in part by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico, Brazil) and the Swiss Government Excellence Scholarships for Foreign Scholars and Artists (ESKAS No. 2015.0478).

References

[1] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia. The totem multiple-ring ordering and topology maintenance protocol. *ACM Trans. Comput. Syst.*, 16(2):93–132, May 1998.

[2] A. Babay and Y. Amir. Fast total ordering for modern data centers. In *ICDCS*, 2016.

[3] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[4] E. Bezerra, D. Cason, and F. Pedone. Ridge: high-throughput, low-latency atomic multicast. In *SRDS*, 2015.

[5] K. Birman and T. Joseph. Reliable communication in the presence of failures. *Trans. on Computer Systems*, 5(1):47–76, Feb. 1987.

[6] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[8] J. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX ATC*, 2012.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[10] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[11] C. Delporte-Gallet and H. Fauconnier. Fault-tolerant genuine atomic multicast to multiple groups. In *OPODIS*, 2000.

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

[13] U. Fritzke, P. Ingels, A. Mostéfaoui, and M. Raynal. Fault-tolerant total order multicast to asynchronous groups. In *SRDS*, 1998.

[14] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, 254(1-2):297–316, 2001.

[15] R. Guerraoui and M. Vukolić. Refined quorum systems. *Distributed Computing*, 23(1):1–42, 2010.

[16] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[17] J. D. J. C. Corbett and M. E. et al. Spanner: Google's globally distributed database. In *OSDI*, 2012.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[19] L. Lamport. The part-time parliament. *Trans. on Computer Systems*, 16(2):133–169, 1998.

[20] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.

[21] P. J. Marandi, M. Primi, and F. Pedone. Multi-ring paxos. In *DSN*, 2012.

[22] F. Pedone. Boosting system performance with optimistic distributed protocols. *Computer*, 34(12):80–86, Dec. 2001.

[23] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *DISC*, 1998.

[24] L. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *IC3N*, 1998.

[25] N. Schiper and F. Pedone. Optimal atomic broadcast and multicast algorithms for wide area networks. In *PODC*, 2007.

[26] N. Schiper and F. Pedone. On the inherent cost of atomic broadcast and multicast in wide area networks. In *ICDCN*, 2008.

[27] N. Schiper, P. Sutra, and F. Pedone. Genuine versus non-genuine atomic multicast protocols for wide area networks: An empirical study. In *SRDS*, 2009.

[28] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[29] D. Sciascia, F. Pedone, and F. Junqueira. Scalable deferred update replication. In *DSN*, 2012.

[30] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[31] P. Zieliński. Optimistic generic broadcast. In *DISC*, 2005.

Appendix

Proofs for Propositions 1 and 2

Proposition 1 *Let Algorithm 1 use the reliable multicast implementation in [6] and the consensus implementation in [19]. Every atomically multicast global message is delivered in at least* 6δ.

PROOF: Let m be a global message a-multicast using Algorithm 1. It takes one communication delay for m to be r-multicast to its destinations. Each group in $m.dst$ relies on consensus to assign m a hard tentative timestamp. Executing an instance of consensus within m's destination groups takes 2δ in the best case [20]. Processes then exchange their timestamps in another communication delay and finally execute one more instance of consensus before a-delivering m. Thus, m can be a-delivered in (at least) 6δ. □

Proposition 2 *Assume Algorithm 2 uses the reliable multicast implementation in [6] and the consensus implementation in [19]. If* $\mathscr{R}$ *is a set of runs of Algorithm 2 in which for every group* g *there is a correct leader* l_g *and for each* $p \in g$ $leader_{g,p} = l_g$*, then there are runs in* $\mathscr{R}$ *in which atomically multicast global messages are delivered fast, in four communication delays.*

PROOF: Consider run $R \in \mathscr{R}$ where a process r a-multicasts a global message m. From Algorithm 2, r r-multicasts $(\text{START}, \perp, \perp, m)$ to $m.dst$. For every g in $m.dst$ and each p in g, p r-delivers m at Task 1 within one communication delay and adds tuple $(\text{SET-HARD}, g, \perp, m)$ to *ToOrder*. We consider now what happens in the fast and in the slow paths.

- In the fast path, from the hypothesis and Task 4, for each g in $m.dst$, process l_g (and no other process in g) assigns a soft timestamp C_s to m and r-multicasts it to $m.dst$ in a SEND-SOFT message, which processes r-deliver after one delay (Task 2) and as a result include tuple $(\text{SYNC-SOFT}, h, y, m)$ in *ToOrder*. Then, l_g in g proposes a sequence that contains the SYNC-SOFT message and after two delays, processes decide on the value proposed and include $(\text{SYNC-SOFT}, h, y, m)$ in $\mathscr{B}$. In total, four communication delays are needed since m is a-multicast.

- In the slow path, after including $(\text{SET-HARD}, g, \perp, m)$ in *ToOrder*, l_g (and no other process in g) proposes a sequence that includes the SET-HARD tuple, which is decided by processes in g after two delays. After deciding on the SET-HARD tuple, processes in g assign hard timestamp C_h to m and r-multicast $(\text{SEND-HARD}, g, C_h, m)$ to all processes in $m.dst$ (Task 5). This message is r-delivered within one delay and upon r-delivering it (Task 3), processes include $(\text{SET-HARD}, g, C_h, m)$ in $\mathscr{B}$. In total, four communication delays since m is a-multicast.

Thus, for each process in m's destination and each group $g \in m.dst$, after four delays tuple $(\text{SYNC-SOFT}, g, x, m)$ is in $\mathscr{B}$ and tuple $(\text{SYNC-HARD}, g, x, m)$ is in *ToOrder* and p includes $(\text{SYNC-HARD}, g, x, m)$ in $\mathscr{B}$ (Task 6) and a-delivers m (Task 7), after four communication delays. □

Proof of correctness for Algorithm 1

Proposition 3 *Uniform integrity: For any process* p *and any message* m*,* p *a-delivers* m *at most once, and only if* $p \in m.dst$ *and* m *was previously a-multicast.*

PROOF: From p to a-deliver m (Task 5), p assessed that for each h in $m.dst$, there is a tuple $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$. We show SYNC-HARD tuples for m are included only once in $\mathscr{B}$. There are two cases in which tuple $(\text{SYNC-HARD}, h, x, m)$ is included in $\mathscr{B}$. In Task 4, $(\text{SYNC-HARD}, h, x, m)$ is included in $\mathscr{B}$ upon deciding on tuple $(\text{SET-HARD}, h, x, m)$ if m is local and upon deciding on $(\text{SYNC-HARD}, h, x, m)$ if m is global. For the first case to happen, $(\text{SET-HARD}, h, x, m)$ must be included in *ToOrder* in Task 1, as the result of the r-delivery of $(\text{START}, \perp, \perp, m)$. From the properties of reliable broadcast, such a message is delivered only once by p. The second case happens when a SYNC-HARD tuple is in *ToOrder*, which from Task 2, it is included only once. From Algorithm 2, it follows immediately that p only a-delivers m if p is part of m's addresses and m is a-multicast. □

Lemma 1 *If a correct process* p *in* g *includes tuple* T *in ToOrder, then eventually processes in* g *decide on a sequence of tuples that contains* T.

PROOF: Process p includes T in *ToOrder* either in Task 1 or in Task 2. In both cases, T was r-delivered by p and from the properties of reliable broadcast, every correct process in g will r-deliver T and include it in *ToOrder*. Let t be a time after which all faulty processes have failed. Thus, after t there is a time when all *ToOrder* sequences from processes that propose in consensus contain T. By the uniform integrity property of consensus, T is eventually included in a decision of consensus. □

Lemma 2 *For each correct process p that has tuple* $(\text{SYNC-HARD}, h, x, m)$ *in $\mathscr{B}$, p eventually replaces the entry by* $(\text{FINAL}, \perp, ts, m)$ *in $\mathscr{B}$ where ts is the maximum timestamp x in the* SYNC-HARD *tuples that concern m.*

PROOF: To include $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$, p has decided on a sequence that contains either
(a) a $(\text{SET-HARD}, h, x, m)$ tuple if m is local, or (b) a $(\text{SYNC-HARD}, h, x, m)$ tuple if m is global. In case (a), $(\text{SYNC-HARD}, h, x, m)$ will be trivially replaced by $(\text{FINAL}, \perp, x, m)$ in Task 5. In case (b), some process proposed a *ToOrder* \ *Ordered* sequence that contains $(\text{SYNC-HARD}, h, x, m)$. The SYNC-HARD tuple is included in *ToOrder* in Task 2 upon r-delivering tuple $(\text{SEND-HARD}, h, x, m)$, which was r-multicast in Task 4, upon the decision of a sequence with $(\text{SET-HARD}, h, x, m)$. Thus, $(\text{SET-HARD}, h, x, m)$ was included in *ToOrder* at Task 1, as a result of the r-delivery of $(\text{START}, \perp, \perp, m)$, which is r-multicast to all of m's destinations. Every group h in $m.dst$ upon r-delivering $(\text{START}, \perp, \perp, m)$ adds tuple $(\text{SET-HARD}, h, x, m)$ to *ToOrder*, which from Lemma 1 is eventually included in a consensus decision and results in the r-multicast of $(\text{SEND-HARD}, h, x, m)$ to members of $m.dst$. When a process r-delivers $(\text{SEND-HARD}, h, x, m)$, it adds $(\text{SYNC-HARD}, h, x, m)$ to *ToOrder* and, from Lemma 1, the tuple is decided in an instance of consensus, leading to the inclusion of $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$. Once there is a tuple $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$ for each group h in $m.dst$, p replaces the SYNC-HARD tuples by $(\text{FINAL}, \perp, ts, m)$. □

Lemma 3 *If a correct process p includes* $(\text{FINAL}, \perp, ts, m)$ *in $\mathscr{B}$, then p eventually a-delivers m.*

PROOF SKETCH: Assume for a contradiction that q does not a-deliver m. Thus, there is some tuple (z, h, y, m') in $\mathscr{B}$ such that $m \neq m'$ and $y < ts$. We first show that eventually any entry (z, h, y, m') added in $\mathscr{B}$ after $(\text{FINAL}, \perp, ts, m)$ is in $\mathscr{B}$ has a timestamp bigger than ts. Message m only has a FINAL tuple in $\mathscr{B}$ after it received SYNC-HARD tuples from all of m's destinations. When q includes $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$ in Task 4, q updates C_h such that it contains the maximum between it current value and x. Since the next SET-HARD event that q handles for a message m'' will increment C_h, it follows that m'' will have a timestamp bigger than ts.

We now show that every message that contains a timestamp smaller than m's final timestamp ts is eventually a-delivered and removed from $\mathscr{B}$. Let (z, h, y, m') be an entry in $\mathscr{B}$ such that $y < ts$. Either z is FINAL or it is SYNC-HARD and from Lemma 2 z the tuple will eventually be replaced by a FINAL tuple. Thus, from Task 5 message m' will be eventually a-delivered and removed from $\mathscr{B}$, a contradiction. We conclude then that q eventually a-delivers m. □

Proposition 4 *Validity: If a correct process p a-multicasts a message m, then eventually all correct processes $q \in m.dst$ a-deliver m.*

PROOF: Upon a-multicasting m, p r-multicasts $(\text{START}, \perp, \perp, m)$ to all processes in $m.dst$ and from the validity and agreement properties of reliable broadcast, every correct q in $m.dst$ will r-deliver $(\text{START}, \perp, \perp, m)$ in Task 1. By Task 1, q includes $(\text{SET-HARD}, g, \perp, m)$ in *ToOrder* and from Lemma 1, the tuple is eventually decided in some instance of consensus. If m is global, q r-multicasts $(\text{SEND-HARD}, g, \perp, m)$ to all processes in $m.dst$; if m is local, q includes $(\text{SYNC-HARD}, g, \perp, m)$ in $\mathscr{B}$. In the first case, every correct process $r \in m.dst$ eventually r-delivers $(\text{SEND-HARD}, g, \perp, m)$ and includes $(\text{SYNC-HARD}, g, \perp, m)$ in *ToOrder*. From Lemma 1 the SYNC-HARD tuple is eventually included in a consensus decision and from Task 4 in $\mathscr{B}$. Therefore, eventually SYNC-HARD tuples from every group in $m.dst$ are in $\mathscr{B}$ and q replaces them by $(\text{FINAL}, \perp, ts, m)$, where ts is the maximum among the timestamps in the SYNC-HARD tuples. From Lemma 3, q eventually a-delivers m. □

Proposition 5 *Uniform agreement: If a process p a-delivers a message m, then eventually all correct processes $q \in m.dst$ a-deliver m.*

PROOF SKETCH: For process p to a-deliver m, from Task 5 for every group h in $m.dst$, there is a tuple $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$. Moreover, there is no entry (z, h', y, m') in $\mathscr{B}$ such that $m \neq m'$ and $ts < y$, where ts is the maximum timestamp in the SYNC-HARD tuples that concern m. We consider two cases.

Case (a): p and q are in the same group g. Process p adds tuples to $\mathscr{B}$ in Task 4 only, and every tuple added to $\mathscr{B}$, which can be of type SET-HARD and SYNC-HARD, is created from an entry in *Decided* and variable C_h. Moreover, C_h is modified only in Task 4, based on tuples in *Decided*. Since p and q decide on the same sequence of consensus values, it follows that unless q fails, it executes the same sequence of steps as p and eventually a-delivers m.

Case (b): p and q are in different groups. To include $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$, p has decided on a sequence that includes the tuple, for each group $h \in m.dst$. From consensus, some process r proposed a sequence that contains $(\text{SYNC-HARD}, h, x, m)$ in *ToOrder* (Task 3). So, r r-delivered tuple $(\text{SEND-HARD}, h, x, m)$, which was r-multicast to all of m's destinations in Task 4. As a consequence, processes in q's group r-deliver message $(\text{SEND-HARD}, h, x, m)$ and decide on a sequence that includes $(\text{SYNC-HARD}, h, x, m)$. We conclude that r adds $(\text{SYNC-HARD}, h, x, m)$ to $\mathscr{B}$, for each of m's destinations h. From Lemma 3, q eventually a-delivers m. □

Proposition 6 *Uniform prefix order: For any two messages m and m' and any two processes p and q such that $\{p,q\} \subseteq m.dst \cap m'.dst$, if p a-delivers m and q a-delivers m', then either p a-delivers m' before m or q a-delivers m before m'.*

PROOF: The proposition trivially holds if p and q are in the same group, so assume p is in group g and q is in group h and suppose, by way of contradiction, that p does not a-deliver m' before m nor does q a-deliver m before m'. Without loss of generality, suppose that $m.ts < m'.ts$.

We claim that q inserts m into $\mathscr{B}$ before a-delivering m'. In order for m (respectively, m') to be a-delivered by p (resp., q), p's (resp., q's) $\mathscr{B}$ must contain tuples $(\text{SYNC-HARD}, g, x, m)$ and $(\text{SYNC-HARD}, h, y, m)$ (resp., $(\text{SYNC-HARD}, g, x', m')$ and $(\text{SYNC-HARD}, h, y', m')$). From Task 4, for p to include a SYNC-HARD tuple in $\mathscr{B}$, p must have decided a sequence that contains $(\text{SYNC-HARD}, g, x, m)$ (recall that m is a global message). Thus, some process in g included $(\text{SYNC-HARD}, g, x, m)$ in *ToOrder*, after r-delivering tuple $(\text{SEND-HARD}, g, x, m)$. With a similar argument, some process in g included $(\text{SYNC-HARD}, g, x', m')$ in *ToOrder*, after r-delivering tuple $(\text{SEND-HARD}, g, x', m')$. Let r and s be the processes that r-multicast messages $(\text{SEND-HARD}, g, x, m)$ and $(\text{SEND-HARD}, g, x', m')$, respectively, at Task 4. Therefore, r and s decided sequences that include the SET-HARD tuples. Assume that $(\text{SET-HARD}, g, x, m)$ is decided before $(\text{SET-HARD}, g, x', m')$. Therefore, before r-multicasting $(\text{SEND-HARD}, g, x', m')$, s r-multicast $(\text{SEND-HARD}, g, x, m)$. From the FIFO properties of reliable multicast, q r-delivered the tuples in the order above and we can show that $(\text{SYNC-HARD}, g, x, m)$ appears in $\mathscr{B}$ before $(\text{SET-HARD}, g, x', m')$, which proves our claim.

Consequently, from the claim, q a-delivers m before m' since $m.ts_q < m'.ts_q$, a contradiction that concludes the proof. □

Proposition 7 *Uniform acyclic order: The relation $<$ is acyclic.*

PROOF SKETCH: Suppose, by way of contradiction, that there exist messages $m_1, ..., m_k$ such that $m_1 < m_2 < ... < m_k < m_1$. From Task 5, processes a-deliver messages following the order of their final timestamps. Thus, there must be processes p and q such that the final timestamps they assign to m_1, $m_1.ts_p$ and $m_1.ts_q$, satisfy $m_1.ts_p < m_1.ts_q$, a contradiction since both p and q receive the same SYNC-HARD tuples used to calculate m_1's final timestamp in Task 5.

□

Theorem 1 *Algorithm 1 implements atomic multicast.*

PROOF: This follows directly from Propositions 3 through 7. □

Proof of correctness for Algorithm 2

Proposition 8 *Uniform integrity: For any process p and any message m, p a-delivers m at most once, and only if $p \in m.dst$ and m was previously a-multicast.*

PROOF: Assume p a-delivers m. From Task 7, p assessed that for each h in $m.dst$, there is a tuple $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$. We show that SYNC-HARD tuples for m are included only once in $\mathscr{B}$. Since SYNC-HARD tuples for m are removed from $\mathscr{B}$ when m is a-delivered (in Task 7), this shows that m is a-delivered only once. There are three cases in which tuple $(\text{SYNC-HARD}, h, x, m)$ is included in $\mathscr{B}$ in Algorithm 2. In Task 5, $(\text{SYNC-HARD}, h, x, m)$ is included in $\mathscr{B}$ (a) upon deciding on tuple $(\text{SET-HARD}, h, x, m)$ if m is local and (b) upon deciding on

(SYNC-HARD, h, x, m) if m is global. (c) In Task 6 if the soft and hard timestamps for h (i.e., (SYNC-SOFT, h, x, m) and (SYNC-HARD, h, x, m)) in *ToOrder* match.

In all cases, after including (SYNC-HARD, h, x, m) in $\mathscr{B}$, p added the tuple to *Ordered* and so, the tuple will not be in any future consensus decision. Moreover, in case (a), (SET-HARD, h, x, m) is included in *ToOrder* upon r-delivering a START message for m (Task 1). From uniform integrity of reliable multicast, p does not r-delivers the START message for m more than once. In cases (b) and (c), since the SYNC-HARD tuple is in *ToOrder*, from Task 3, no other SYNC-HARD tuple for m from h will be included in *ToOrder*.

From Algorithm 2, it follows immediately that p only a-delivers m if p is part of m's addresses and m is a-multicast. □

Lemma 4 *If a correct process p in g includes tuple T in ToOrder, then eventually processes in g decide on a sequence of tuples that contains T.*

PROOF: Process p includes T in *ToOrder* in Tasks 1, 2 and 3. In all cases, T was r-delivered by p and from the properties of reliable broadcast, every correct process in g will r-deliver T and include it in *ToOrder*. Let t be a time after which all faulty processes have failed and t' the time after which for every process p in g, $leader_{g,p} = l_g$, where l_g is a correct process. Thus, after $t'' = max(t, t')$ there is a time when all *ToOrder* sequences contain T and one correct process, l_g, proposes this sequence in consensus. By the uniform integrity property of consensus, T is eventually included in a decision of consensus. □

Lemma 5 *For each correct process p that has tuple (SYNC-HARD, h, x, m) in $\mathscr{B}$, p eventually replaces the entry by (FINAL, $\perp, ts, m$) in $\mathscr{B}$ where ts is the maximum timestamp x in the SYNC-HARD tuples that concern m.*

PROOF: To include (SYNC-HARD, h, x, m) in $\mathscr{B}$, (a) p has decided on a sequence that contains a (SET-HARD, h, x, m) tuple if m is local (Task 5), or (b) p has decided on a sequence that contains a (SYNC-HARD, h, x, m) tuple if m is global (Task 5), or (c) p has decided on a sequence that contains (SYNC-SOFT, h, x, m) and p has included (SYNC-HARD, h, x, m) in *ToOrder*, if m is global (Task 6).

In case (a), (SYNC-HARD, h, x, m) in $\mathscr{B}$ will be trivially replaced by (FINAL, $\perp, x, m$) in Task 5. In case (b), some process proposed a *ToOrder* \ *Ordered* sequence that contains (SYNC-HARD, h, x, m). So, for cases (b) and (c), the SYNC-HARD tuple is included in *ToOrder* in Task 3 upon r-delivering tuple (SEND-HARD, h, x, m), which was r-multicast in Task 5, upon the decision of a sequence with (SET-HARD, h, x, m). Therefore, (SET-HARD, h, x, m) was included in *ToOrder* at Task 1, as a result of the r-delivery of (START, $\perp, \perp, m$), which is r-multicast to all of m's destinations. Every group h in $m.dst$ upon r-delivering (START, $\perp, \perp, m$) adds tuple (SET-HARD, h, x, m) to *ToOrder*, which from Lemma 4 is eventually included in a consensus decision and results in the r-multicast of (SEND-HARD, h, x, m) to members of $m.dst$. When a process r-delivers (SEND-HARD, h, x, m), it adds (SYNC-HARD, h, x, m) to *ToOrder* and, from Lemma 4, the tuple is decided in an instance of consensus, leading to the inclusion of (SYNC-HARD, h, x, m) in $\mathscr{B}$.

Once for each group h in $m.dst$ there is a tuple (SYNC-HARD, h, x, m) in $\mathscr{B}$, p replaces the SYNC-HARD tuples by (FINAL, $\perp, ts, m$). □

Lemma 6 *If a correct process p includes (FINAL, $\perp, ts, m$) in $\mathscr{B}$, then p eventually a-delivers m.*

PROOF SKETCH: Assume for a contradiction that q does not a-deliver m. Thus, there is some tuple (z, h, y, m') in $\mathscr{B}$ such that $m \neq m'$ and $y < ts$.

We first show that eventually any entry (z, h, w, m') added in $\mathscr{B}$ after (FINAL, $\perp, ts, m$) is in $\mathscr{B}$ has a timestamp bigger than ts. Process q only includes (FINAL, $\perp, ts, m$) in $\mathscr{B}$ (Task 7) after from each $h \in m.dst$ (a) q decided on a sequence with (SYNC-HARD, h, x, m) (Task 5) or (b) q decided on a sequence with (SYNC-SOFT, h, x, m) (Task 5) and included (SYNC-HARD, h, x, m) in *ToOrder*, after r-delivering (SEND-HARD, h, x, m) (Task 3). Before q includes (SYNC-HARD, h, x, m) in $\mathscr{B}$ in Tasks 5 and 6, q updates C_h such that it contains the maximum between its current value and x. Since the next SET-HARD event that q handles for a message m'' will increment C_h, m'' will have a timestamp bigger than ts.

We now show that every message that contains a timestamp smaller than m's final timestamp ts is eventually a-delivered and removed from $\mathscr{B}$. Let (z, h, y, m') be an entry in $\mathscr{B}$ such that $y < ts$. Either z is FINAL or it is SYNC-HARD and from Lemma 5 the tuple will eventually be replaced by a FINAL tuple. Thus, from Task 5 message m' will be eventually a-delivered and removed from $\mathscr{B}$, a contradiction. We conclude then that q eventually a-delivers m. □

Proposition 9 *Validity: If a correct process p a-multicasts a message m, then eventually all correct processes $q \in m.dst$ a-deliver m.*

PROOF: Upon a-multicasting m, p r-multicasts $(\textsc{start}, \perp, \perp, m)$ to all processes in $m.dst$ and from the validity and agreement properties of reliable broadcast, every correct q in $m.dst$ will r-deliver $(\textsc{start}, \perp, \perp, m)$ in Task 1. By Task 1, q includes $(\textsc{set-hard}, g, \perp, m)$ in *ToOrder* and from Lemma 4, the tuple is eventually decided in some instance of consensus. If m is global, q r-multicasts $(\textsc{send-hard}, g, \perp, m)$ to all processes in $m.dst$; if m is local, q includes $(\textsc{sync-hard}, g, \perp, m)$ in $\mathscr{B}$. In the first case, every correct process $r \in m.dst$ eventually r-delivers $(\textsc{send-hard}, g, \perp, m)$ and includes $(\textsc{sync-hard}, g, \perp, m)$ in *ToOrder*. From Lemma 4 the SYNC-HARD tuple is eventually included in a consensus decision and from Task 5 in $\mathscr{B}$. Therefore, eventually SYNC-HARD tuples from every group in $m.dst$ are in $\mathscr{B}$ and q replaces them by $(\textsc{final}, \perp, ts, m)$, where ts is the maximum among the timestamps in the SYNC-HARD tuples. From Lemma 6, q eventually a-delivers m. □

Proposition 10 *Uniform agreement: If a process p a-delivers a message m, then eventually all correct processes $q \in m.dst$ a-deliver m.*

PROOF: For process p to a-deliver m, from Task 7, for every group h in $m.dst$, there is a tuple $(\textsc{sync-hard}, h, x, m)$ in $\mathscr{B}$. Process p includes $(\textsc{sync-hard}, h, x, m)$ in $\mathscr{B}$ in the following situations: (a) after p decides on a sequence that contains $(\textsc{set-hard}, h, x, m)$ and m is local (Task 5); (b) after p decides on a sequence that includes $(\textsc{sync-hard}, h, x, m)$ and m is global (Task 5); and (c) after p decides on a sequence that includes $(\textsc{sync-soft}, h, x, m)$ and $(\textsc{sync-hard}, h, x, m)$ is in *ToOrder* (Task 6).

In case (a), from consensus's uniform agreement property, q decides on a sequence that contains $(\textsc{set-hard}, h, x, m)$ and since m is local, q includes $(\textsc{sync-hard}, h, x, m)$ in $\mathscr{B}$. From Lemma 6, q eventually a-delivers m. In case (b), following a similar argument as item (a), q decides on a sequence that includes $(\textsc{sync-hard}, h, x, m)$ and includes the tuple in $\mathscr{B}$. From Lemma 6, q eventually a-delivers m.

In case (c), to include $(\textsc{sync-hard}, h, x, m)$ in *ToOrder*, p has r-delivered message $(\textsc{send-hard}, h, x, m)$ (Task 3), which was r-multicast to all of m's destinations in Task 5 upon the decision of a sequence containing tuple $(\textsc{set-hard}, h, x, m)$ at group h. From consensus's uniform agreement property, every correct process in h also decides on a sequence with $(\textsc{set-hard}, h, x, m)$ and r-multicasts $(\textsc{send-hard}, h, x, m)$ to every process in $m.dst$. Upon r-delivering $(\textsc{send-hard}, h, x, m)$, processes in q's group include tuple $(\textsc{sync-hard}, h, x, m)$ in *ToOrder* and from Lemma 4 decide on a sequence that includes the SYNC-HARD tuple. Then, q includes $(\textsc{sync-hard}, h, x, m)$ in $\mathscr{B}$ and from Lemma 6, q eventually a-delivers m. □

Proposition 11 *Uniform prefix order: For any two messages m and m' and any two processes p and q such that $\{p, q\} \subseteq m.dst \cap m'.dst$, if p a-delivers m and q a-delivers m', then either p a-delivers m' before m or q a-delivers m before m'.*

PROOF: The proposition trivially holds if p and q are in the same group, so assume p is in group g and q is in group h and suppose, by way of contradiction, that p does not a-deliver m' before m nor does q a-deliver m before m'. Without loss of generality, suppose that $m.ts < m'.ts$.

We claim that q inserts m into $\mathscr{B}$ before a-delivering m'. In order for m (respectively, m') to be a-delivered by p (resp., q), p's (resp., q's) $\mathscr{B}$ must contain tuples $(\textsc{sync-hard}, g, x, m)$ and $(\textsc{sync-hard}, h, y, m)$ (resp., $(\textsc{sync-hard}, g, x', m')$ and $(\textsc{sync-hard}, h, y', m')$).

For p to include a SYNC-HARD tuple in $\mathscr{B}$, p must have (a) decided a sequence that contains $(\textsc{sync-hard}, g, x, m)$ (recall that m is a global message) (Task 5) or (b) decided a sequence that contains $(\textsc{sync-soft}, g, x, m)$ and $(\textsc{sync-hard}, h, x, m)$ is in *ToOrder* (Task 6).

In case (a), some process in g included $(\textsc{sync-hard}, g, x, m)$ in *ToOrder*, after r-delivering tuple $(\textsc{send-hard}, g, x, m)$. With a similar argument, some process in g included $(\textsc{sync-hard}, g, x', m')$ in *ToOrder*, after r-delivering tuple $(\textsc{send-hard}, g, x', m')$. In cases (a) and (b), let r and s be the processes that r-multicast messages $(\textsc{send-hard}, g, x, m)$ and $(\textsc{send-hard}, g, x', m')$, respectively, at Task 4. Therefore, r and s decided sequences that include the SET-HARD tuples. Assume that $(\textsc{set-hard}, g, x, m)$ is decided before $(\textsc{set-hard}, g, x', m')$. Therefore, before r-multicasting $(\textsc{send-hard}, g, x', m')$, s r-multicast $(\textsc{send-hard}, g, x, m)$. From the FIFO properties of reliable multicast, q r-delivered the tuples in the order above and we can show that $(\textsc{sync-hard}, g, x, m)$ appears in $\mathscr{B}$ before $(\textsc{set-hard}, g, x', m')$, which proves our claim.

Consequently, from the claim, q a-delivers m before m' since $m.ts_q < m'.ts_q$, a contradiction that concludes the proof. □

Proposition 12 *Uniform acyclic order: The relation $<$ is acyclic.*

PROOF SKETCH: Suppose, by way of contradiction, that there exist messages $m_1, ..., m_k$ such that $m_1 < m_2 < ... < m_k < m_1$. From Task 7, processes a-deliver messages following the order of their final timestamps. Thus, there must be processes p and q such that the final timestamps they assign to m_1, $m_1.ts_p$ and $m_1.ts_q$, satisfy $m_1.ts_p < m_1.ts_q$, a contradiction since both p and q receive the same SYNC-HARD tuples used to calculate m_1's final timestamp in Task 7. □

Theorem 2 *Algorithm 2 implements atomic multicast.*

PROOF: This follows directly from Propositions 8 through 12. □