PrimCast: A Latency-Efficient Atomic Multicast

Leandro Pacheco
Università della Svizzera italiana
Lugano, Switzerland
le.pacheco@gmail.com

Paulo Coelho
Federal University of Uberlândia
Uberlândia, Brazil
paulocoelho@ufu.br

Fernando Pedone
Università della Svizzera italiana
Lugano, Switzerland
fernando.pedone@usi.ch

ABSTRACT

Atomic multicast is a communication abstraction that allows for messages to be addressed to and reliably delivered by multiple process groups, while ensuring a partial order on delivered messages. Strong ordering guarantees can greatly simplify the design and implementation of distributed applications. One critical property for the performance and scalability of an atomic multicast protocol is that of genuineness: a protocol is said to be genuine if only the sender and destinations of a message are involved in ordering the message. This paper presents PrimCast, the first genuine atomic multicast protocol able to deliver messages at every destination in three communication steps. PrimCast uses a primary-based consensus protocol for deciding on message timestamps at each group. Differently from previous work, it does not rely on consensus for advancing and maintaining logical clocks. PrimCast introduces a novel approach, relying on simple quorum intersection, to decide when a multicast message can be delivered. We also show how loosely synchronized clocks can be used to reduce the convoy effect that delays messages under high system load. We present the complete algorithm for PrimCast and evaluate its performance under various scenarios. Our results show that PrimCast achieves lower latency than state-of-the-art approaches while providing higher or comparable throughput.

CCS CONCEPTS

• **Software and its engineering → Consistency**; **Software fault tolerance**; • **Computing methodologies → Distributed algorithms**.

KEYWORDS

atomic multicast, distributed agreement, fault-tolerant distributed systems

ACM Reference Format:
Leandro Pacheco, Paulo Coelho, and Fernando Pedone. 2023. PrimCast: A Latency-Efficient Atomic Multicast. In *24th International Middleware Conference (Middleware '23), December 11–15, 2023, Bologna, Italy.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3590140.3629110

Middleware '23, December 11–15, 2023, Bologna, Italy

ACM ISBN 979-8-4007-0177-1/23/12…$15.00
https://doi.org/10.1145/3590140.3629110

1 INTRODUCTION

Distributed systems use replication to tolerate the failure of system components. In order to scale performance, replicated systems typically rely on data partitioning, also known as sharding [12]. Commonly, in partitioned systems, replicas are divided into groups with each group responsible for storing a subset of the application data. Given a workload where commands are evenly distributed across partitions and most commands access one or a few partitions, system throughput should scale with the number of partitions.

Building distributed applications is a complex endeavor, and data partitioning introduces additional complexity from coordinating operations across replica groups. Atomic multicast is a communication abstraction that simplifies reasoning about and building partitioned systems. It allows for messages to be reliably delivered to a subset of the system groups while ensuring a partial order on delivered messages. Partial ordering is at the core of strongly consistent partitioned systems. While solutions such as [12, 13] rely on ad-hoc timestamping schemes to ensure a valid partial ordering of operations, atomic multicast can be used as a building block for implementing consistent cross-partition operations in distributed databases [18, 39] and replicated applications in general [4, 7, 27, 33]. To be effective, however, an atomic multicast protocol must scale with the number of groups and introduce minimum overhead. One critical property for the scalability of such algorithms is that of genuineness [23], where only the sender and destinations of a message need to coordinate to order the message. Even though non-genuine ordering protocols such as [3, 28] can provide high message throughput, they do not allow for locality of ordering. Local messages, destined to a single group, may have to be delayed by a round-trip to a sequencer. This is particularly relevant in deployments with geographically disperse groups [33]. Most proposals for genuine atomic multicast rely on a timestamping scheme first proposed in Skeen's protocol [8]. Under high load, timestamp-based atomic multicast protocols may exhibit a *convoy effect* [2], where a message's delivery is delayed due to not-yet-delivered messages with a smaller final timestamp.

In this paper, we present PrimCast, the first genuine atomic multicast protocol able to deliver messages to every destination in three communication steps. Like previous approaches to atomic multicast [11, 19, 20, 23, 37], PrimCast relies on a timestamping scheme first proposed in Skeen's protocol [8]: Each process group maintains a logical clock and uses it to assign a *local timestamp* to each message destined to it. A message's *final timestamp* is computed as the maximum of all its local timestamps. Processes deliver messages in final timestamp order to ensure a partial ordering of deliveries across the whole system. PrimCast uses a primary-based consensus protocol for deciding on local timestamps at each group. Each destination process individually tracks quorums for each message, allowing local timestamps to be known at every destination in

three message delays. Differently from previous work [11, 20, 21], PrimCast does not rely on consensus for advancing and maintaining logical clocks. Processes inside each group exchange clock values and rely on simple quorum intersection to decide when a message can be safely delivered, that is, when no future message may be assigned a smaller final timestamp.

While application semantics can be used to reorder the delivery of messages that do not conflict [34], we propose a new technique to reduce the effects of convoy that does not depend on message contents. In an approach similar to hybrid logical clocks [26], we make use of loosely synchronized clocks when proposing message timestamps to significantly reduce the latency introduced by the convoy effect. Clock synchronization, however, is not required for the correctness of the protocol.

Besides presenting the complete algorithm for PrimCast, we have built a prototype and compared its performance against state-of-the-art protocols (i.e., [11, 20]). Our results show that PrimCast consistently delivers lower latency than state-of-the-art protocols while providing higher (up to 4x as high in some cases) maximum throughput than the alternatives. The results also demonstrate how our proposed solution of using loosely synchronized clocks can in some situations almost eliminate the convoy effect in PrimCast.

The rest of the paper is organized as follows. Sections 2 and 3 introduce the system model and definitions. Section 4 surveys related work. Sections 5 and 6 present PrimCast and the loosely synchronized clocks approach, respectively. Section 7 describes our experimental evaluation. Section 8 concludes the paper.

2 MODEL AND DEFINITIONS

In this section, we detail our system model and recall the definitions of reliable and atomic multicast.

2.1 System model

We assume a distributed system composed of a finite set of interconnected processes. There is an unbounded set of *client processes* and a bounded set of *server processes* Π. Processes may fail by crashing, but do not experience arbitrary or malicious behavior (i.e., no Byzantine failures). A process that crashes is said to be *faulty*, otherwise it is *correct*. Processes communicate by message passing through pairwise communication channels. Channels do not create, corrupt or duplicate messages, and given two correct processes p and q, if p sends m to q, q eventually receives m.

We define $\Gamma = \{g_1, g_2, ..., g_m\}$ as the set of process groups in the system. Process groups are disjoint [23], and $\bigcup \Gamma = \Pi$. Associated with each group g, there is a set of quorums Q_g. Each quorum q in Q_g is a set of processes, such that $q \subset g$. The intersection between any two quorums in Q_g cannot be empty, and at least one of the quorums in Q_g must contain no faulty processes.

We consider a system that is *partially synchronous* [15]: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *global stabilization time* (GST) and is unknown to the processes. Before the GST, there are no bounds on the time it takes for messages to be transmitted and actions to be executed. After the GST, such bounds exist but are unknown, and remain in effect forever. In practice, "forever" means long enough for atomic multicast to make progress, that is, deliver messages.

For simplicity, we consider that processes in group g have access to a *weak leader election oracle*, Ω_g, which can be implemented in a partially synchronous system [1]. At each process in the group, Ω_g outputs a single process contained in g and has the following property: there is a time after which, at every correct $p_i \in g$, Ω_g outputs the same correct process $p_l \in g$.

2.2 Reliable and Atomic Multicast

For every message m, $m.dest$ denotes the groups to which m is multicast. If $|m.dest| = 1$ we say that m is a *local* message; if $|m.dest| > 1$ 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 consider non-uniform FIFO reliable multicast, which ensures the following properties:

- *Validity:* If a correct process executes R-MULTICAST(m) then, eventually, all correct processes in $\bigcup m.dest$ execute R-DELIVER(m).
- *Integrity:* For any message m and process p, p may only do R-DELIVER(m) once, and only if R-MULTICAST(m) was previously issued by some process.
- *Non-uniform agreement:* If a correct process executes R-DELIVER(m) then, eventually, every correct process in $\bigcup m.dest$ executes R-DELIVER(m).
- *FIFO order:* If a process executes R-MULTICAST(m) before executing R-MULTICAST(m') then, every process that executes R-DELIVER(m') must first execute R-DELIVER(m).

In FIFO non-uniform reliable multicast, messages multicast and delivered by faulty processes may be lost (i.e., may never be delivered by correct processes), which allows for implementations that deliver messages in *one communication step*, from origin to destinations [24].

Atomic multicast provides two primitives to processes in the system: A-MULTICAST(m) to send messages and A-DELIVER(m) to signal deliveries. Atomic multicast satisfies the uniform integrity and validity properties of reliable multicast as well as the following properties:

- *Uniform agreement:* If any process executes A-DELIVER(m) then, eventually, every correct process in $\bigcup m.dest$ executes A-DELIVER(m).
- *Global total order:* Let $\prec$ be a relation on the set of messages that processes A-DELIVER, such that $m \prec m'$ iff some process executes A-DELIVER(m) before it executes A-DELIVER(m'). The $\prec$ relation is acyclic.
- *Uniform prefix order:* Let m and m' be messages and p and q processes such that $\{p, q\} \subseteq \bigcup(m.dest \cap m'.dest)$. If p executes A-DELIVER(m) and q executes A-DELIVER(m') then either p executes A-DELIVER(m') before A-DELIVER(m) or q executes A-DELIVER(m) before A-DELIVER(m').

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

A multicast protocol is *genuine* [23] when only the sender and destinations of a message m need to take steps for m to be delivered. Intuitively, a genuine protocol scales with the number of groups in

the system, as long as most messages are local or destined to only a subset of the groups. We formally define genuineness as follows:

- *Genuineness:* for any admissible run $\mathcal{R}$ of the algorithm and for any process p, if p sends or receives a message in $\mathcal{R}$ then m is a-multicast in $\mathcal{R}$ and either $p \in \bigcup m.dest$ or p does the a-multicast of m.

In [23], the authors show the impossibility of solving genuine atomic multicast with weak synchronous assumptions when groups intersect. Hence, we assume that groups are disjoint. We note that this limitation is not a big issue in practice, as processes from different groups can be collocated on the same physical machine.

3 BACKGROUND

In this section, we provide an overview of Skeen's protocol, upon which PrimCast and other timestamp-based atomic multicast solutions are based (§3.1). We also describe the notions of collision-free and failure-free latencies, which we use to characterize the delivery latency of PrimCast (§3.2).

3.1 Timestamp-based message ordering

Most atomic multicast protocols achieve a partial ordering of messages through a timestamping scheme first proposed in what is known as Skeen's protocol [8]. Skeen's protocol works as follows:

(1) Each process in the system has its own logical clock.
(2) A message m, destined to processes in $m.dest$, is sent to each one of these processes.
(3) A process that receives m increments its logical clock and uses the clock to assign m a *local timestamp*. The local timestamp is sent to other processes in $m.dest$ and m becomes a pending message at p.
(4) Once a process receives a local timestamp from each process in $m.dest$, the maximum of the local timestamps is chosen as m's *final timestamp*. The process then updates its clock to m's final timestamp, if not already past it.
(5) Process p delivers message m once no pending messages at p have a possibly smaller final timestamp than m's.

Skeen's protocol is genuine, as only the sender and processes in $m.dest$ take steps to deliver m. In [19], Fritzke et al. propose a solution for fault-tolerant atomic multicast. This solution is further refined in [37]. The core idea is to replace individual processes as destinations with fault-tolerant process groups. Inside each group, atomic broadcast (i.e., consensus) is used to both maintain the group's logical clock and to timestamp messages. We refer to protocols that rely on assigning message timestamps as timestamp-based.

3.2 Collision-free and failure-free latency

In [20], Gotsman et al. propose two metrics for describing the delivery latency of atomic multicast protocols: failure-free and collision-free latency. We consider the *delivery latency* of a message as the time between its a-multicast and its last a-delivery, that is, the time for the message to be a-delivered at every correct destination.[1] Both collision-free and failure-free latencies set bounds on the delivery latency of messages in periods of system stability. More precisely, we consider the system stable after some unknown time t, occurring past GST (§2.1), when message delay is bounded, leaders at each group are stable (i.e., the output of Ω won't change) and there is no ongoing or future reconfiguration due to previous leader changes. For simplicity, we assume that local computation takes no time and message delay between any two processes is fixed after t (a communication step).

Collision-free latency is the maximum delivery latency for a message when there are no conflicting concurrent messages. A message m is *concurrent* with another message m' if m is a-multicast before m' is first a-delivered, and m' is a-multicast before m is first a-delivered. Two messages m and m' are *conflicting* iff $m.dest \cap m'.dest \neq \emptyset$. In the presence of concurrent messages, message delivery may be subject to a *convoy effect* [2, 9]. In timestamping protocols, messages need to be delivered in final timestamp order. Thus, a given message m can only be delivered when there is no other message, yet to be delivered, with a possibly smaller final timestamp than m's. Effectively, m can potentially have its delivery delayed by any message that is multicast at a time before m is assigned a local timestamp in each of the groups in $m.dest$. This effect is more relevant in a wide-area deployment, with high latency between groups. In such a scenario, the time it takes for messages to get their local and final timestamps can vary considerably, depending on the location of the sender and the destinations of the message. Under high load, there is a high probability that messages are assigned local timestamps in such a way that a message that could otherwise be delivered, has to wait for a newer message to get its final timestamp. *Failure-free latency* is the maximum delivery latency for a message in the presence of concurrent messages, that is, when considering the worst case convoy effect. In practice, in periods of system stability, the failure-free and collision-free latencies can be seen as the worst and best case delivery latencies of an atomic multicast protocol, respectively. For a detailed analysis of the convoy effect in atomic multicast we refer to [2].

A method is proposed in [20] to calculate the collision-free and failure-free latency values in timestamp-based atomic multicast protocols. First, two values must be obtained from the algorithm: the *clock update latency* C and the *commit latency* D. Let m be a message multicast at time t, as defined above. The clock update latency C is the maximum delay after which no group in $m.dest$ will assign another message a local timestamp smaller than m's final timestamp. Essentially, the clock update latency limits the interval in which conflicting concurrent messages can be a-multicast. The commit latency D of m is the maximum delay after which a destination knows the final timestamp of m and has its group's logical clock value equal to or higher than m's final timestamp. In the absence of conflicting concurrent messages, m can be delivered after $t + D$, and the collision-free latency is thus equal to D. The earliest a concurrent message can be multicast before m is $t + C$. Thus, after $t+C+D$, every final timestamp smaller than m's must be known, and the respective messages can be delivered together with m. Hence, the failure-free latency is equal to $C + D$.

4 RELATED WORK

Most proposals for genuine atomic multicast are derived from the timestamping scheme of Skeen's protocol [8]. In this section, we

[1] This differs from the definition in [20], which considers the first a-delivery of a message.

give a detailed account of the FastCast (§4.1) and White-Box (§4.2) atomic multicast protocols and provide an overview of other related protocols (§4.3).

4.1 FastCast

In [11], Coelho et al. propose FastCast, a genuine atomic multicast protocol with collision-free and failure-free latency values of 4 and 8 communication steps, respectively. FastCast achieves faster delivery times than the classic protocols through an optimistic execution path that works as follows:

(1) Each group has an elected leader, responsible for proposing local timestamps at its group.
(2) When timestamping m, a leader sends its proposal to the leaders of every other group in $m.dest$, before proposing the timestamp through consensus in its group.
(3) Once a leader gets proposals from all leaders in $m.dest$, it sends the maximum of all proposals as an optimistic final timestamp through its group's consensus.

If the final timestamp matches the optimistic timestamp, the message can be delivered before the second sequential round of consensus (which is still executed if necessary). The optimistic path can be understood as a mechanism for updating a group's logical clock before the final timestamp is decided. In FastCast, both the commit latency and the clock update latency are equal to 4 communication steps. Hence, collision-free and failure-free latency values are 4 and 8 communication steps respectively.

4.2 White-Box multicast

In [20], Gotsman et al. propose White-Box, an atomic multicast protocol that improves the collision-free and failure-free latency values to 4 and 6 communication steps, respectively. Furthermore, at group leaders, delivery happens one step earlier, in 3 and 5 communication steps. Differently from previous approaches, White-Box does not use a consensus protocol as a black-box, opting instead for an integrated protocol at group and global level. Another insight from White-Box is the use of a *primary-based* approach (i.e., passive replication) [25, 30]. The primaries at each group decide on the order of messages and then ensure the other replicas follow that same order.

The White-Box protocol works roughly as follows:

(1) A process a-multicasts m by sending a message to the primaries of each group in $m.dest$.
(2) Each primary then picks a local timestamp for m for its group, based on its own clock value. It then sends that proposal to every process in every group in $m.dest$, as an ACCEPT message.
(3) Once a process receives the ACCEPT from every primary in $m.dest$, it will store the proposal for its group and update its clock to the highest local timestamp received, if needed. It then sends back an ACK to each of the primaries in $m.dest$.
(4) After receiving all the ACCEPTs and a quorum of ACKs from each group in $m.dest$, a primary picks the largest local timestamp of m as m's final timestamp. Primaries carefully track pending messages to a-deliver messages in final timestamp order. At a primary, m can be a-delivered in as early as 3 communication steps from being a-multicast. Then, a DELIVER message is sent to other processes in the group.
(5) Followers a-deliver messages in the order of the DELIVER messages sent by the primary, in as early as 4 communication steps from the respective a-multicast.

From the above, we get to the collision-free latency of 4 communication steps, or 3 communication steps, when only considering delivery at the primaries. Once a message m is a-multicast, after 2 communication steps (enough for primaries in $m.dest$ to exchange local timestamp proposals and update their clocks), no new conflicting message can be assigned a local timestamp smaller than the final timestamp of m, as long as primaries remain stable. Hence, the failure-free latency of White-Box is 6 communication steps, or 5 when considering delivery at primaries only.

4.3 Other protocols

[23] and [19] propose the use of multiple instances of consensus, one per group, to solve atomic multicast when processes may fail. These protocols can deliver a message in 6 communication steps in the collision-free case, that is, when there are no concurrent messages. These two protocols have a failure-free delivery latency of 12 communication steps.

In [37], Schiper et al. propose improvements to [19] that reduce the collision-free latency at some of the destination groups, those that assign a local timestamp equal to the final timestamp of the message, to 3 communication delays.

In MTO [22] and Scalatom [36], instead of running multiple consensus instances per message (on per group), a single instance of consensus is run among all destination groups. These protocols achieve a collision-free and failure-free latency values of 5 and 9, respectively.

Tempo [16] is a partitioned state-machine replication protocol that is built over a protocol that is essentially a genuine atomic multicast implementation. Instead of relying on a primary at each destination group, each message has a designated leader in each group (the closest replica) that communicates with other processes in the group to assign the message a local timestamp. To decide when messages are safe for execution, Tempo uses a notion of timestamp stability that works in parallel with the timestamping of messages, similar to how PrimCast exchanges bump messages to update quorum-clock() values. Even though the two approaches have similarities, PrimCast and Tempo have been developed in parallel.

Many non-genuine solutions to atomic multicast have also been proposed in the literature. In [38], Schiper et al. propose a round based protocol which can deliver messages in 4 communication steps. An unbounded sequence of rounds is executed, and each group chooses a set of messages to be delivered at each round. Proposals in the each round are exchanged and then deterministically ordered and delivered by the destinations. ByzCast [10] is a byzantine fault-tolerant atomic multicast that organizes process groups in a tree. Each message is first ordered by the lowest common ancestor of its destinations, and then proceeds down the tree being ordered by each group until it is ordered at each destination. Partial order is ensured by having each group respect the ordering of its ancestors.

In Multi-Ring Paxos [31], Ridge [6] and Elastic Paxos [5], processes subscribe to the groups they are interested in receiving messages from, and then a deterministic merge procedure is used ensure a partial ordering of messages. These protocols have a slightly different interface: a message can only destined to a single group, but groups don't have to be disjoint: sending a message to multiple groups requires a group that is a superset of those destinations. For an overview of total and partial order communication abstractions we refer to [14].

PrimCast relies on a primary-based approach (i.e., passive replication) for deciding on timestamps inside each group [25, 30]. Since the primary orders all operations in a group, it can optimistically update its state before having the rest of the group agree on it. This property can be exploited for faster logical clock updates inside groups. We refer to [42] for a discussion of the differences between state-machine replication and primary-based replication.

5 PRIMCAST

In this section, we discuss PrimCast's basic ideas and then present the algorithm in detail. Due to lack of space, proof of correctness for the properties of the protocol can be found in [32].

5.1 Basic ideas

PrimCast is a genuine atomic multicast protocol that achieves collision-free and failure-free latency of three and five communication steps, respectively, *at every destination*. This is a reduction of one communication step from the state-of-the-art, which achieved these latency values only at group leaders [20].

PrimCast is based on the following ideas:

- *Primary-based consensus at each group:* Each group in PrimCast employs a primary-based consensus procotol. Similarly to other primary-based protocols [25, 30], PrimCast is epoch based. Each epoch is owned by a single process in the group. Inside a group, each process tracks its current epoch, only accepting proposals from the primary of that epoch. In the absence of failures, when processes in a group follow the same epoch, advancing the logical clock of the primary to a given value is enough to ensure new messages are assigned a larger local timestamp. Hence, for any message m, after two communication steps (i.e., the time for group primaries to exchange their timestamp proposals), no other message can be assigned a local timestamp smaller than the final timestamp of m. The clock update latency C is thus two communication steps.
- *Quorum-based logical clocks:* One of the requirements for a message m to be safely delivered at a given destination is that no new messages targeting the same destination should be assigned a smaller final timestamp than m's. Updating the logical clock of primaries is enough to prevent this situation in the failure-free case, but when the primary changes, this is not enough, as we now explain. To ensure safety in the presence of failures, previous approaches rely on consensus to agree on the group's logical clock, and delivery of a message m can only happen at a given process after its group's logical clock is larger than or equal to m's final timestamp. In PrimCast, instead of relying on consensus for logical clock agreement, a quorum-based approach is used. Inside each group, processes track each other's clock values. On an epoch change, the new primary must pick a clock value larger than all values seen in a quorum of clocks from previous epochs. When a message m is multicast, by carefully exchanging and tracking clock values, every destination can have its group's logical clock advanced past m's final timestamp in three communication steps.
- *Cross-group quorum tracking:* Instead of exchanging local timestamps after consensus is reached inside each group, PrimCast replicas directly send acknowledgment messages to other destination groups. Each destination process individually tracks when the quorum for a local timestamp from another group is reached. Every local timestamp for a given message is thus learned in three communication steps at every destination. This, together with quorum-based logical clocks, ensures the commit latency D is three communication steps at every destination.

5.2 Algorithm

PrimCast is presented in Algorithm 1 (initialization and predicates), Algorithm 2 (main logic), and Algorithm 3 (primary change logic). Processes communicate through the *r-multicast* and *r-deliver* primitives of FIFO non-uniform reliable broadcast (§2.2), which can deliver messages in one communication step [24].

In the following, we give an overview of the algorithm and provide some insights into how it achieves safety.

5.2.1 A note on epochs. PrimCast employs a primary-based protocol inside each group to assign local timestamps to messages. The protocol proceeds in epochs, a given epoch $\mathcal{E}$ being owned by a single process p, the epoch leader. If a quorum of processes accept $\mathcal{E}$ as their *current epoch* ($\mathcal{E}_{cur} = \mathcal{E}$), p may become the effective primary. Epochs from different groups are not related: each group has its own set of epochs, and advances epochs independently of other groups.

5.2.2 Assigning timestamps. To a-multicast a message m, the sender r-multicasts $\langle \text{START}, m \rangle$ to each destination in $\bigcup m.dest$ (line 31). When r-delivered, the tuple is added to the $\mathcal{M}$ set. The primary for each group in $m.dest$ will eventually update its *clock*, pick a timestamp for m, append the proposal to $\mathcal{T}$ and send the respective ACK to every destination in $m.dest$ (line 35).

When a process $p \in g$ r-delivers an ACK for m coming from a process in its own group g (line 40), p first stores the tuple in $\mathcal{M}$. Then, if the ACK is coming from the primary of its current epoch, p accepts the timestamp proposal by appending it to $\mathcal{T}$, updates its *clock* if needed, and then also r-multicasts its own ACK to every destination in $m.dest$. When p instead r-delivers an ACK for m coming from a process in a remote group h (i.e., $p \notin h$), p simply stores the tuple in $\mathcal{M}$ (line 46).

The local timestamp of m for h is tracked by local-ts(m, h) (line 9). The value is decided when, in $\mathcal{M}$, there are ACKs for m from a quorum of processes in h, all coming from the same epoch. The final-ts(m) is decided once local-ts(m, g) is decided for every $g \in m.dest$. When primaries are stable, after three communication steps every

Algorithm 1 PrimCast initialization and definitions at process $p \in g$.

```
1:  initialization:
2:      M ← ∅                                          ▷ set of r-delivered START, ACK and BUMP tuples
3:      D ← ∅                                          ▷ set of a-delivered messages
4:      T ← ∅                                          ▷ sequence of tuples for timestamps proposed in g (in the format ⟨E, m, ts⟩)
5:      clock ← 0                                      ▷ p's clock value
6:      E_cur ← initial epoch                          ▷ current epoch
7:      E_prom ← initial epoch                         ▷ promised epoch (always ≥ E_cur)
8:      state ← PRIMARY if leader(E_cur) = p else FOLLOWER
9:  local-ts(m, h) ≡                                   ▷ local timestamp for m in h if known, otherwise ⊥
10:     if ∃ts, E', quorum ∈ Q_h : ∀q ∈ quorum : ⟨ACK, m, h, E', ts, q⟩ ∈ M then ts
11:     else ⊥
12: final-ts(m) ≡                                     ▷ max of all local-ts in m.dest if all are decided, otherwise ⊥
13:     if ∀h ∈ m.dest : local-ts(m, h) ≠ ⊥ then max_{h∈m.dest}(local-ts(m, h))
14:     else ⊥
15: min-clock(q) ≡                                    ▷ highest ts seen in messages from q in epoch E_cur or earlier
16:     max({0} ∪ {ts | ∃E' ≤ E_cur : ⟨ACK, _, g, E', ts, q⟩ ∈ M or ⟨BUMP, E', ts, q⟩ ∈ M})
17: quorum-clock() ≡                                  ▷ lower bound for clock of the primary of epochs higher than E_cur
18:     max({ts | ∃quorum ∈ Q_g : ∀q ∈ quorum : min-clock(q) ≥ ts})
19: min-ts(m) ≡                                       ▷ minimum possible value for final-ts(m)
20:     max(if ∃h : local-ts(m, h) ≠ ⊥ then max_{h∈m.dest}(local-ts(m, h)) else 0,
21:         min(if ∃E, ts : ⟨E, m, ts⟩ ∈ T then ts else ∞,          ▷ any local-ts is a lower bound, so
22:             1 + min-clock(leader(E_cur)),                       ▷ is the minimum possible proposal for m in g
23:             1 + quorum-clock())))
24: proposable(m) ≡                                   ▷ m is not decided or proposed in g
25:     ⟨START, m⟩ ∈ M and local-ts(m, g) = ⊥ and ⟨_, m, _⟩ ∉ T
26: deliverable(m) ≡
27:     m ∉ D and final-ts(m) ≠ ⊥ and                 ▷ m has not been delivered and has a final timestamp
28:     final-ts(m) ≤ min-clock(leader(E_cur)) and    ▷ smaller than new proposals in E_cur
29:     final-ts(m) ≤ quorum-clock() and              ▷ and smaller than proposals in newer epochs
30:     ∀m' : ⟨_, m', _⟩ ∈ T, m' ∉ D, m' ≠ m : ⟨final-ts(m), m.id⟩ < ⟨min-ts(m'), m'.id⟩
                                                      ▷ and smaller than the possible timestamp of any other pending m'
```

correct destination in $\bigcup m.dest$ will have received an ACK for m from every other correct destination, ensuring final-ts(m) is decided.

5.2.3 Delivering a message. A process can only safely deliver a message m when (1) the process knows final-ts(m), (2) every message with a smaller final timestamp has been delivered, and (3) no message may yet be assigned a smaller final timestamp. At a given process $p \in g$, these conditions are tracked by the deliverable(m) predicate (line 26), with message ids used to break ties. This predicate depends on the following definitions:

- final-ts(m) (line 12): the final timestamp of m is known once the local-ts(m, h) (i.e., the local timestamp) for all $h \in m.dest$ are known.
- min-clock(q) (line 15): the maximum clock value seen in messages from process q, from epochs smaller or equal to $\mathcal{E}_{cur}$.
- quorum-clock() (line 17): lower bound for the starting clock of primaries for epochs higher than $\mathcal{E}_{cur}$ in the process's group. For a process p to become the primary in its group g, p must first obtain a quorum of promises for the new epoch from processes in g (line 65). The largest clock value seen in the set of received promises is chosen as the starting clock value of the new epoch (line 68). As an example, consider a group g of 5 processes with simple majority quorums (i.e., any 3 processes is a quorum). Suppose a new leader p starts an epoch $\mathcal{E}$, gets a promise from each process in g (including itself), and the set of clock values gathered from the promises is $\{1, 2, 3, 4, 5\}$. The minimum clock value that can be picked by p for the new epoch is 3, which comes from the quorum of promises with values $\{1, 2, 3\}$. From quorum intersection, there is a quorum of promises for which all clock values ($\{3, 4, 5\}$) are higher than or equal to 3. Thus, processes rely on quorum-clock() to know when a given timestamp is safe for delivery in epochs higher than $\mathcal{E}_{cur}$. For this reason, min-clock(q) ignores tuples coming from epochs higher than $\mathcal{E}_{cur}$.
- min-ts(m) (line 19): lower bound for the final timestamp of message m. Any known local timestamp for m is a lower bound. At process $p \in g$, when local-ts(m, g) is not yet known, a lower bound can be inferred for its future value.

Algorithm 2 PrimCast algorithm at process $p \in g$.

31: **a-multicast**(m): ▷ process p wants to atomically multicast m to $m.dest$
32: r-multicast($\langle$START, $m\rangle$) to $m.dest$

33: **when** r-deliver($\langle$START, $m\rangle$):
34: $\mathcal{M} \leftarrow \mathcal{M} \cup \{\langle$START, $m\rangle\}$

35: **when** $\exists m$: proposable(m) **and** $state$ = PRIMARY: ▷ primary proposes local timestamp in g
36: **for each** m : proposable(m)
37: $clock \leftarrow clock + 1$
38: $\mathcal{T} \leftarrow \mathcal{T} \bullet \langle \mathcal{E}_{cur}, m, clock\rangle$
39: r-multicast($\langle$ACK, $m, g, \mathcal{E}_{cur}, clock, p\rangle$) to $m.dest$

40: **when** r-deliver($\langle$ACK, $m, h, \mathcal{E}, ts, q\rangle$) **and** $g = h$: ▷ on ACK from our group
41: $\mathcal{M} \leftarrow \mathcal{M} \cup \{\langle$ACK, $m, h, \mathcal{E}, ts, q\rangle\}$
42: **if** q = leader($\mathcal{E}$) **and** $\mathcal{E} = \mathcal{E}_{cur}$ **and** $state$ = FOLLOWER **then** ▷ if ACK from primary
43: $\mathcal{T} \leftarrow \mathcal{T} \bullet \langle \mathcal{E}_{cur}, m, ts\rangle$ ▷ send our own ack
44: $clock \leftarrow \max(clock, ts)$
45: r-multicast($\langle$ACK, $m, g, \mathcal{E}_{cur}, ts, p\rangle$) to $m.dest$

46: **when** r-deliver($\langle$ACK, $m, h, \mathcal{E}, ts, q\rangle$) **and** $g \neq h$: ▷ on ACK from remote group
47: $\mathcal{M} \leftarrow \mathcal{M} \cup \{\langle$ACK, $m, h, \mathcal{E}, ts, q\rangle, \langle$START, $m\rangle\}$
48: **if** $ts > clock$ **then** ▷ on a remote ACK with ts higher than our $clock$
49: $clock \leftarrow ts$ ▷ update $clock$ and inform our group
50: r-multicast($\langle$BUMP, $\mathcal{E}_{prom}, clock, p\rangle$) to g

51: **when** r-deliver($\langle$BUMP, $\mathcal{E}, ts, q\rangle$):
52: $\mathcal{M} \leftarrow \mathcal{M} \cup \{\langle$BUMP, $\mathcal{E}, ts, q\rangle\}$

53: **when** $\exists m$: deliverable(m) **and** $state \in \{$PRIMARY, FOLLOWER$\}$:
54: **for each** m : deliverable(m)
55: $\mathcal{D} \leftarrow \mathcal{D} \cup \{m\}$
56: a-deliver(m) ▷ deliver m to the application

The value of local-ts(m, g) will come either from the current primary (equal to the proposal in $\mathcal{T}$ or higher than 1+min-clock(leader($\mathcal{E}_{cur}$))) or from the primary of some future epoch (higher than quorum-clock()).

5.2.4 Propagating clock values inside a group. Instead of relying on consensus to maintain a group's logical clock, PrimCast carefully tracks the clock values from processes in the group, and uses quorum intersection to ensure safety during epoch changes (see the explanation for quorum-clock() in the previous section). Clock values are propagated in two ways: (1) implicitly through ACK messages or (2) through BUMP messages. Whenever p receives an ACK from q, it will update its own clock if needed. The ACK also updates what p knows about q's clock value. Processes in g will exchange ACKs for the local timestamp of m in g among themselves. This exchange is enough to both (1) move the clocks of processes in g past the local timestamp for m in g and (2) inform processes about the updated clock values. For a message m to be delivered, a process must know that a quorum of clocks in its group is past the final timestamp of m. When the local timestamp for some remote group is the largest for m, the ACKs alone are not enough to ensure delivery. Thus, when an ACK from a remote group is received, a process updates its clock if needed and sends a BUMP message to its group (line 50). Note that BUMP messages carry the sender's promised epoch, $\mathcal{E}_{prom}$: once a process is promised to epoch $\mathcal{E}$ it cannot influence the quorum-clock() calculation for epochs lower than $\mathcal{E}$.

5.2.5 Example execution. Figure 1 shows some of the messages sent during an example execution of the protocol. In the example we have two groups, $g = \{p_1, p_2, p_3\}$ and $h = \{p_4, p_5, p_6\}$, and we consider simple majority quorums for each. Processes p_1 and p_4 are the primaries of $\mathcal{E}_g$ and $\mathcal{E}_h$ respectively. Process p_5 does a-multicast(m), where $m.dest = \{g, h\}$. The diagram only shows the messages needed for process p_2 to a-deliver m. As it can be seen, in the absence of concurrent messages, m can be a-delivered by p_2 in 3 communication steps. The example also shows why BUMP messages are needed. Without the BUMP messages in $\mathcal{M}$, the value of quorum-clock() at p_2 would be equal to 1, preventing m from being delivered since final-ts(m) = 2.

5.2.6 Changing a group's primary. When a process $p \in g$ has $\Omega_g = p$, if p is not already the leader (line 57), p starts an epoch change. The new leader p starts by picking an epoch $\mathcal{E}$ higher than the one it is promised to ($\mathcal{E}_{prom}$). It then becomes a CANDIDATE, sending a NEW-EPOCH message to processes in g. Any process that receives the NEW-EPOCH, if $\mathcal{E}$ is larger than its promised epoch, becomes PROMISED to $\mathcal{E}$ (line 61) and then sends its current state ($\mathcal{E}_{cur}$, $\mathcal{T}$ and $clock$) to p. Once p gets a quorum of promises for $\mathcal{E}$ (line 65), it must pick the most up-to-date state from the promises

Algorithm 3 PrimCast primary change algorithm at process $p \in g$.

57: **when** $\Omega_g = p$ **and** $state \notin \{\text{PRIMARY}, \text{CANDIDATE}\}$:
58: $state \leftarrow$ CANDIDATE
59: $\mathcal{E}_{prom} \leftarrow$ next epoch higher than $\mathcal{E}_{prom}$ for which p is the leader
60: r-multicast($\langle$NEW-EPOCH, $\mathcal{E}_{prom}\rangle$) to g

61: **when** r-deliver($\langle$NEW-EPOCH, $\mathcal{E}\rangle$) **and** $\mathcal{E} \geq \mathcal{E}_{prom}$:
62: **if** $p \neq$ leader($\mathcal{E}$) **then** $state \leftarrow$ PROMISED
63: $\mathcal{E}_{prom} \leftarrow \mathcal{E}$
64: r-multicast($\langle$PROMISE, $\mathcal{E}, p, clock, \mathcal{E}_{cur}, \mathcal{T}\rangle$) to p

65: **when** $state =$ CANDIDATE **and** PROMISES for $\mathcal{E}_{prom}$ from a $quorum \in Q_g$ were r-delivered:
66: $\mathcal{E}_{max} \leftarrow$ highest epoch in promises
67: $\mathcal{T}_{max} \leftarrow$ longest state from promises with $\mathcal{E}_{max}$ ▷ get $\mathcal{T}$ from most up-to-date replica
68: $ts \leftarrow$ maximum clock from promises ▷ see predicate quorum-clock
69: r-multicast($\langle$NEW-STATE, $\mathcal{E}_{prom}, \mathcal{T}_{max}, ts\rangle$)

70: **when** r-deliver($\langle$NEW-STATE, $\mathcal{E}, \mathcal{T}', ts\rangle$) **and** $\mathcal{E} = \mathcal{E}_{prom}$:
71: $\mathcal{T} \leftarrow \mathcal{T}'$
72: $\mathcal{E}_{cur} \leftarrow \mathcal{E}$ ▷ move p to $\mathcal{E}_{cur}$
73: $clock \leftarrow \max(clock, ts)$
74: r-multicast($\langle$ACCEPT, $\mathcal{E}_{cur}, p\rangle$) to g ▷ inform other replicas we're at $\mathcal{E}_{cur}$

75: **when** $state \in \{\text{PROMISED}, \text{CANDIDATE}\}$ **and** $\mathcal{E}_{cur} = \mathcal{E}_{prom}$ **and** ▷ when p is at $\mathcal{E}_{cur}$ and
76: ACCEPTS for $\mathcal{E}_{cur}$ from some $quorum \in Q_g$ were r-delivered: ▷ so is a $quorum \in Q_g$
77: **if** $state =$ PROMISED **then** $state \leftarrow$ FOLLOWER
78: **if** $state =$ CANDIDATE **then** $state \leftarrow$ PRIMARY
79: **for each** $\langle \mathcal{E}, m, ts\rangle$ **in** $\mathcal{T}$
80: **if** $\langle$ACK, $m, g, \mathcal{E}, ts, p\rangle \notin \mathcal{M}$ **then** ▷ send ACKs we have not yet sent (in $\mathcal{T}$'s order)
81: r-multicast($\langle$ACK, $m, g, \mathcal{E}, ts, p\rangle \notin \mathcal{M}$) to $m.dest$

received: out of all the promises with the highest current epoch, it picks the longest $\mathcal{T}$. Then, p picks the highest clock value out every promise as the starting clock value of the new epoch $\mathcal{E}$. Before p becomes the primary, it must ensure that the new epoch state is safe in a quorum of processes in g. Thus, p sends a NEW-STATE message to all processes in g (line 69). When a process receives the NEW-STATE for the epoch it is promised to (line 70), it installs the state by setting $\mathcal{T}$ and $\mathcal{E}_{cur}$, updates its clock, and then sends an ACCEPT message to every process in g. Once a process has $\mathcal{E}_{cur} = \mathcal{E}$ and receives a quorum of ACCEPTS for $\mathcal{E}$ (line 76), it either becomes the PRIMARY (in case of p) or a FOLLOWER. Finally, for each tuple present in $\mathcal{T}$, in order, if the process has not yet sent the respective ACK (i.e., the ack is not present in $\mathcal{M}$), it sends it to the relevant destinations (line 80).

5.2.7 On liveness. Eventually, from the properties of the leader election oracle Ω_g and our model assumptions (§2), at each group g, the same correct process $p \in g$ is forever output by Ω_g at every process in g. If p is not the primary of g then, from the algorithm (line 57), it will start a new epoch and eventually become the primary. Any message m destined to g that is a-multicast by a correct processes, if not yet proposed or delivered in g (lines 24 and 35), will eventually be present in p's $\mathcal{M}$ set and be proposed by p. Since no other process in g starts a new epoch, and p is correct, m is eventually assigned a local timestamp in g. The same is true for each other group in $m.dest$ and for messages with a smaller timestamp than m in g, thus m is eventually assigned a final timestamp and delivered at g. In practice, it is enough that primaries at each group are stable for periods long enough for local timestamps to be decided and propagated to other groups.

6 EXPLOITING LOOSELY SYNCHRONIZED CLOCKS

Many datacenters today provide loosely synchronized clocks through the use of satellite and atomic clocks [12, 40]. When synchronized clocks are available, we propose the following modification to PrimCast, inspired by hybrid logical clocks [26]. Assuming that real-clock() returns the server's hardware clock value, we modify line 37 as follows:

$clock \leftarrow \max(clock + 1, \text{real-clock}())$

Assume Δ to be the communication step latency, and that clocks are synchronized with a maximum skew of ϵ from real time (i.e., 2ϵ from each other). By having the primary update its clock before proposing a message's local timestamp, the failure-free delivery latency changes from 5Δ to $\min(5\Delta, 4\Delta + 2\epsilon)$. The argument is as follows. Let m be a message delivered at process $p \in g$ with final timestamp ts, and let t be the time at which m is a-multicast. Since it takes at most Δ for m to arrive at any primary in $m.dest$ from being a-multicast, the maximum timestamp possibly assigned to m is $t + \Delta + \epsilon$. Let m' be the message with largest local timestamp at g smaller than ts. The latest time at which m' can be a-multicast and still be ordered before m by some primary in $m.dest$ is the minimum of:

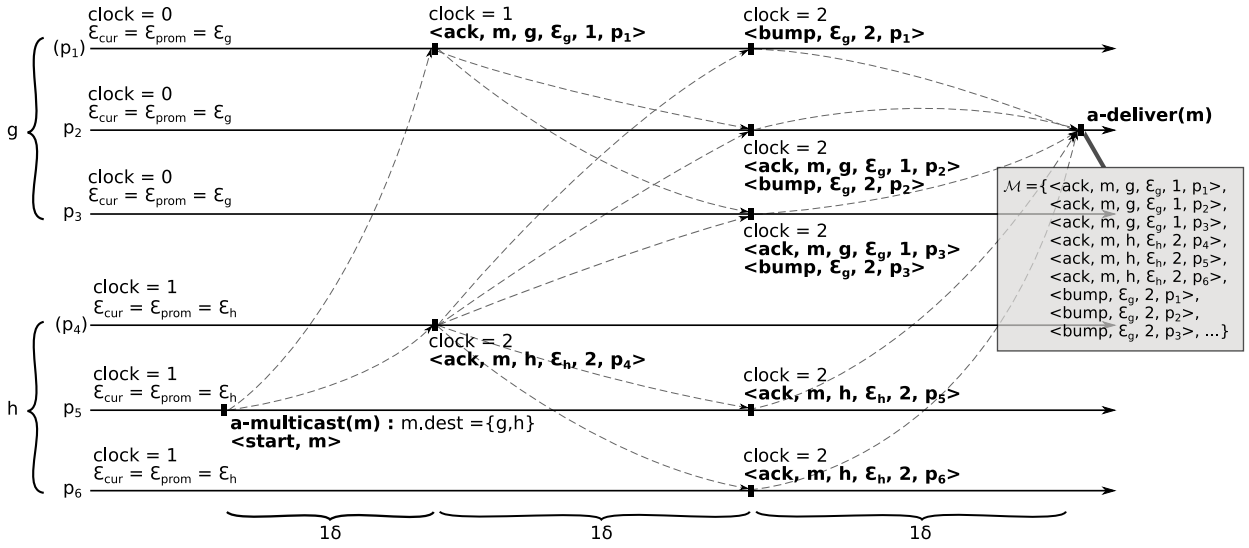


Figure 1: Example execution of PrimCast, showing only messages needed for p_2 to a-deliver a message a-multicast by p_5.

- t+2Δ: the maximum time for primaries in $m.dest$ to exchange proposals and update their clock to ts.
- $t + \Delta + 2\epsilon$: the time for m to reach primaries is $t + \Delta$, and 2ϵ comes from the worst case of ts being assigned a value ϵ in the future and ts' a value ϵ in the past.

Since the collision-free latency of PrimCast is 3Δ, the final timestamp of m' is known at p, at the latest, by time $t+min(2\Delta, \Delta+2\epsilon)+3\Delta$, allowing for m to be delivered. Assuming 2ϵ is smaller than Δ, this effectively reduces the worst case convoy effect by the difference between the two. We show in §7.5 that this technique is particularly effective in a geographically distributed deployment where 2ϵ can be roughly an order of magnitude smaller than Δ [12]. We note that this modification does not impact the correctness of the algorithm, and also cannot increase the worst case convoy effect, even if clocks are not synchronized.

7 PERFORMANCE EVALUATION

In this section, we experimentally evaluate PrimCast. Our evaluation has the objectives of (1) showing how PrimCast compares against the state-of-the-art protocols, under varying system load, (2) how PrimCast behaves under the worst-case scenario, where all messages are destined to every group, and (3) showing the impact of the convoy effect in the different protocols.

7.1 Implementation

We implemented a prototype of PrimCast in Rust using Tokio [41], an asynchronous runtime for building network applications. The prototype is available at [35]. Our implementation is not specifically designed for multi-thread execution, but Tokio's executor can exploit the parallelism and we run PrimCast with 2 threads. In our experiments, we compare PrimCast against two state-of-the-art atomic multicast protocols, FastCast and White-Box (see §4 for details). Table 1 summarizes the main characteristics of the three protocols. For both FastCast and White-Box, we use the open-source implementations provided by the authors in [17] and [43] respectively, both implemented in C using libevent [29]. We use in-memory storage for all implementations. We also run PrimCast using the hybrid clock approach described §6 (PrimCast HC in the figures).

In our implementation, processes rely on TCP for FIFO ordering, and use reconnections and timeouts to decide when to request missing information from other processes. We note this is true for the other implementations as well. Furthermore, while PrimCast exhibits a quadratic communication pattern, a lot of the information exchanged between processes is redundant and needs to be received only once. Our implementation includes optimizations such as sending a message's payload to each replica only once (with the START tuple) and merging sequential acknowledgements into a single message when possible.

7.2 Setup and scenarios

We run all experiments in a cluster, each machine consisting of an eight-core Intel Xeon L5420 2.5GHz processor, 8GB of memory, and 1Gbps ethernet card. The RTT (round-trip time) inside the cluster is around 0.09ms. Besides the deployment inside a LAN, we consider two different emulated wide-area network (WAN) scenarios. To emulate WAN latencies, we used Linux traffic control tools. In all scenarios we deploy 8 groups, each group consisting of 3 replicas. Table 2 summarizes the deployment scenarios. The scenario with colocated leaders evaluates the performance of the protocols in a WAN deployment when group leaders are colocated in the same datacenter. We emulate 3 geographic regions, each with one datacenter, and deploy one replica from each group in each datacenter. We use the latency values reported in [20], with a standard deviation of 5%. To evaluate the convoy effect in the different protocols, we also emulate a WAN deployment with distributed leaders. Each of the 8 groups is deployed to its own geographic region, the RTT being

Protocol	Collision-free latency	Failure-free latency	Message complexity for a-multicast to k groups of size n
FastCast	4	8	$kn + 2k^2n + 2kn + 2kn^2$ $k(2kn + 3n + 2n^2)$ (start) + (snd-soft + snd-hard) + (2× paxos 2a) + (2× paxos 2b)
White-Box	3 (at leaders) 4	5 (at leaders) 6	$k + k^2n + k^2n + kn$ $k(1 + 2kn + n)$ (start) + (leaders accept) + (followers ack) + (deliver)
PrimCast	3	5	$kn + k^2n + k^2n^2 + kn^2$ $k(kn + kn^2 + n + n^2)$ (start) + (leaders ack) + (followers ack) + (bump*)

Table 1: Protocol latency and message complexity. *Bump messages not always required.

Scenario	Cross-group RTT (between leaders)	Intra-group RTT	Description
LAN	0.09ms	0.09ms	8 groups deployed inside a cluster.
WAN - colocated leaders	0.09ms	60ms, 76ms, 130ms	3 regions, each of the 8 groups deployed across them.
WAN - distributed leaders	90ms	30ms	8 regions, each with 3 datacenters. Each group deployed in its own region.

Table 2: Deployment scenarios

90ms between regions and 30ms inside a region, with a standard deviation of 5%.

We collocate one client with each replica in the system. For each message, a client chooses the destination groups at random, except for the group of the replica it is connected to, which is always included. To increase the system load, we uniformly increase the number of outstanding messages from each client. Latency is measured at the client as the time from the message being sent to it being delivered and returned to the client by its replica. We report latency values gathered from all clients in the system.

7.3 LAN performance

Figure 2 compares the performance of the four protocols in a cluster deployment, as load increases, with every message multicast to 2 destinations. Our results show that PrimCast has better performance than both FastCast and White-Box, at every load level measured. FastCast reaches saturation earlier, as it needs to run a slow and a fast path in parallel for message delivery (§4.1). When compared to White-Box, even though PrimCast relies on a quadratic communication pattern, the extra data that needs to be exchanged consists mostly of small acknowledgment information, allowing for an efficient implementation, as our results show. We also note that the hybrid clock approach does not have any particular impact on performance when leaders are colocated, as the convoy effect is mostly a function of cross-group latency. Even tough none of the protocols were designed with a LAN deployment in mind, these results show that PrimCast can be a reasonable alternative in a LAN.

7.4 WAN with colocated leaders

Figure 3 shows how the three protocols behave under increasing load, with messages multicast to 1, 2, 4 or 8 destination groups.

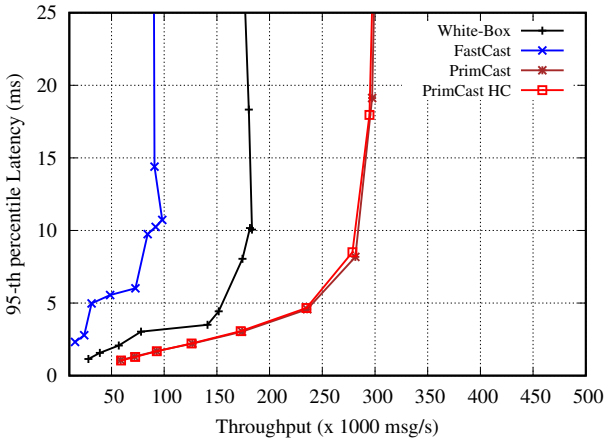
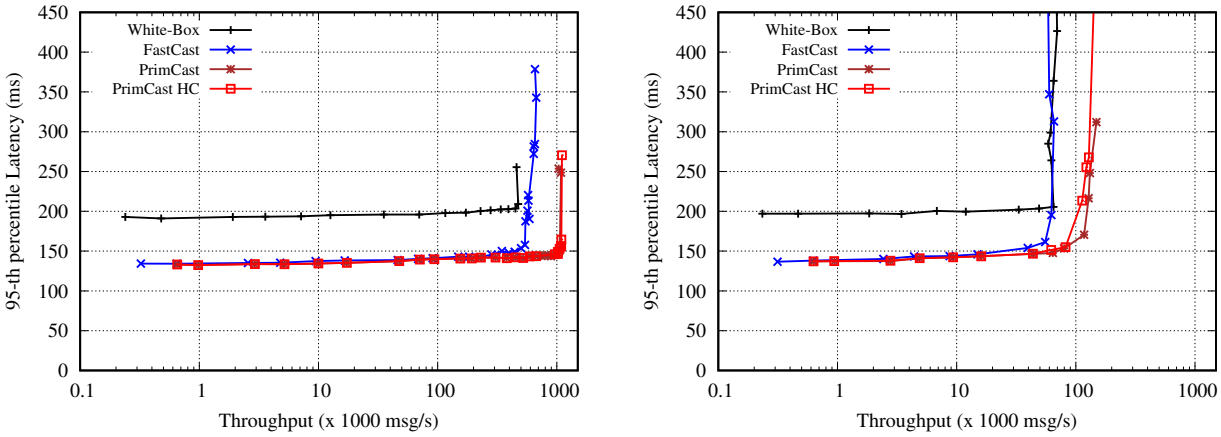


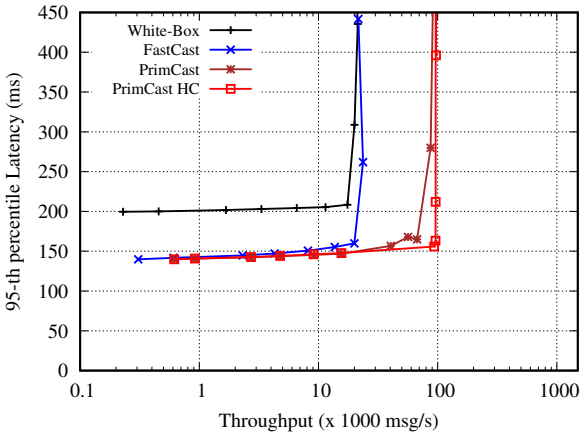
Figure 2: Throughput and 95th-percentile latency in a LAN, with all messages multicast to two groups.

Both PrimCast and FastCast exhibit the same latency behaviour until close to saturation. In the common setup of 3 replicas per group, FastCast can also quickly deliver messages at non-leader replicas. Even then, PrimCast can deliver from 1.6x (1 destination) to 5x (2 destinations) the throughput of FastCast. While some of this difference can be be accounted for by the use of 2 threads in PrimCast's asynchronous execution library, FastCast performance degrades faster with increasing destinations due to the fast and slow paths that need to be executed by the protocol. White-Box on the other hand, needs one extra communication step from leaders, where a message is initially delivered, to the other replicas. This extra latency shows in the 95th percentile latency over all

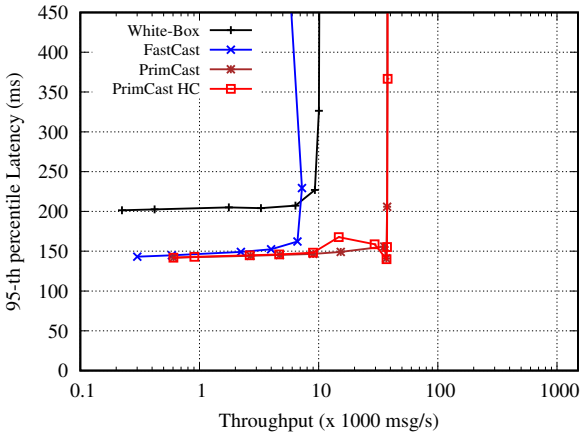


(a) Messages destined to a single group.

(b) Messages destined to 2 groups.

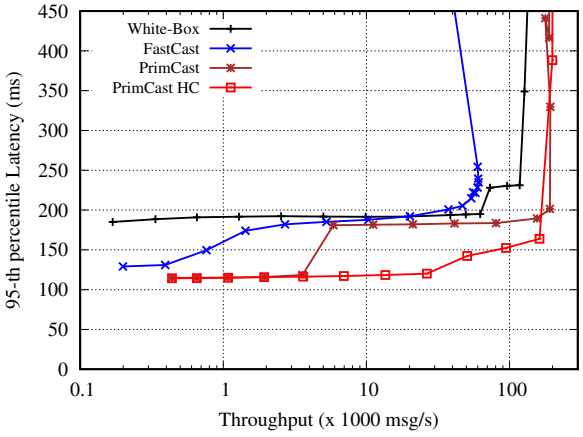


(c) Messages destined to 4 groups.

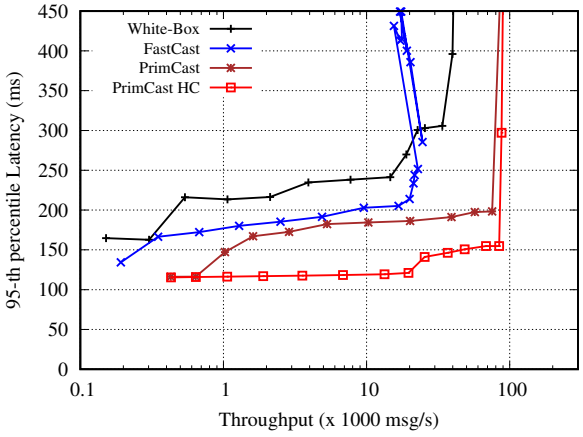


(d) Messages destined to all 8 groups.

Figure 3: Throughput and 95th latency in a WAN with no cross-group latency (i.e., collocated leaders).

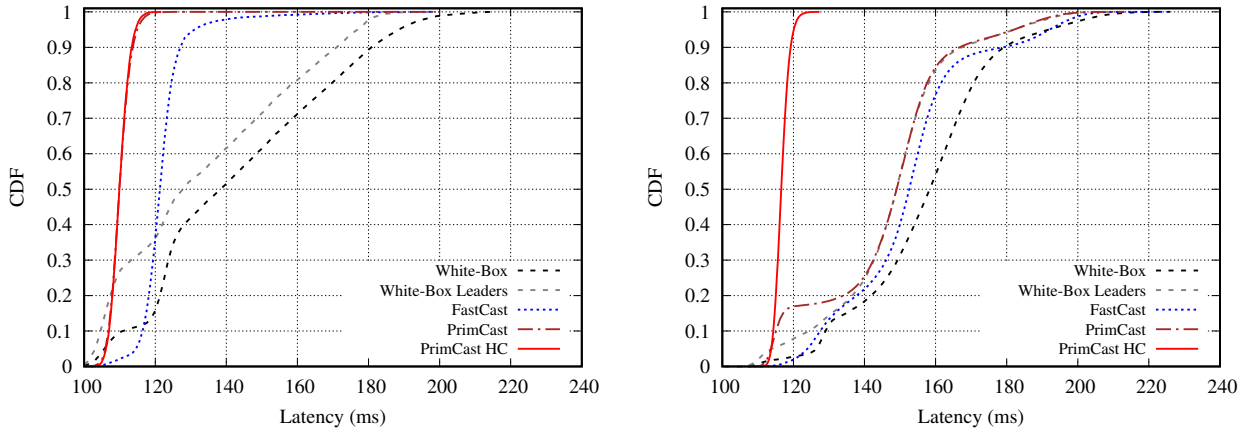


(a) Messages destined to 2 groups.



(b) Messages destined to 4 groups.

Figure 4: Throughput and 95th latency in a WAN with high cross-group latency.



(a) 2 destination groups, 2 outstanding msgs per client (low convoy) **(b) 2 destination groups, 128 outstanding msgs per client (high convoy)**

Figure 5: Latency CDFs at two different load levels, corresponding to the 2nd and 8th points from the curves in Figure 4a

clients. Similarly to the results in a LAN, the convoy effect in this deployment is almost non-existent, as it is a function of cross-group latency. Hence, using hybrid clocks has no effect on latency.

7.5 WAN with distributed leaders

Figure 4 shows the behaviour of the three protocols with messages multicast to 2 and 4 destination groups, as load increases. Differently from the previous deployment, the convoy effect is now clearly visible. From the latency curves, it can be seen that the convoy effect kicks in at different load levels in each protocol, but all are affected by it. As with the previous deployment, White-Box shows worse 95th latency due to extra delay needed to deliver messages at non-leader replicas. Furthermore, PrimCast is able to deliver messages at every destination earlier than any of the two other protocols: exactly one intra-group communication step earlier, around 15ms in this deployment. Moreover, in this setup, using hybrid clocks greatly reduces the latency induced by the convoy effect.

Figure 5 shows the latency distribution for all clients in the system, for each protocol, at two different system loads: one with low load and thus low convoy (left), and one with high load (right). Figure 5a demonstrates how PrimCast consistently delivers lower latencies at every replica in the system. Figure 5b shows how the convoy effect impacts most messages in the system once it takes effect, and also how using hybrid clocks can almost eliminate the effects of convoy in this particular workload. For White-Box, we also isolate the latency values for messages multicast and delivered at group leaders. In a deployment with distributed leaders, White-Box seems heavily affected by the convoy effect even at low load levels, and PrimCast has lower delivery latencies even when only considering deliveries at group leaders. We explain this phenomenon as follows. In White-Box, both leaders and followers need to wait for quorums before forwarding information to other processes. In PrimCast, both acks and clock updates are exchanged immediately, with quorums only checked when delivering a message.

8 CONCLUSION

This paper presented PrimCast, a genuine atomic multicast protocol that can deliver messages, from sender to any destination process, in 3 communication steps. In the presence of conflicting messages, delivery happens at every destination in at most 5 communication steps. This is an improvement over previous work, which needed at least 4 (or 6 in the presence of concurrency) communication steps for delivery at some of the destinations. PrimCast achieves lower latency through the usage of a primary-based replication mechanism and a novel way of tracking logical clocks through simple quorum intersection. We also describe how to exploit loosely synchronized clocks to reduce the impact of the convoy effect that happens under high load. Our experimental evaluation of PrimCast shows that it consistently delivers lower latency than the alternatives while still providing higher throughput. The results also show that, in some cases, using loosely synchronized clocks can almost eliminate the effects of convoy on delivery latency.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Jérémie Decouchant, for their valuable feedback. This work was partially supported by the Swiss National Science Foundation (# 175717), and Conselho Nacional de Desenvolvimento Científico e Tecnológico—CNPq Universal project 407139/2021-4.

REFERENCES

[1] Marcos K Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. 2001. Stable leader election. In *Distributed Computing: 15th International Conference, DISC 2001 Lisbon, Portugal, October 3–5, 2001 Proceedings 15*. Springer, 108–122.

[2] Tarek Ahmed-Nacer, Pierre Sutra, and Denis Conan. 2016. The convoy effect in atomic multicast. In *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*. IEEE, 67–72.

[3] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. 2012. Corfu: A shared log design for flash clusters. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 1–14.

[4] Samuel Benz, Parisa Jalili Marandi, Fernando Pedone, and Benoît Garbinato. 2014. Building Global and Scalable Systems with Atomic Multicast. In *15th ACM/IFIP/USENIX International Middleware Conference (Middleware)*.

[5] Samuel Benz and Fernando Pedone. 2017. Elastic Paxos: A Dynamic Atomic Multicast Protocol. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2157–2164.

[6] Carlos Eduardo Bezerra, Daniel Cason, and Fernando Pedone. 2015. Ridge: high-throughput, low-latency atomic multicast. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 256–265.

[7] Carlos Eduardo Bezerra, Fernando Pedone, and Robbert Van Renesse. 2014. Scalable state-machine replication. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 331–342.

[8] Kenneth P Birman and Thomas A Joseph. 1987. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)* 5, 1 (1987), 47–76.

[9] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. 1979. The convoy phenomenon. *ACM SIGOPS Operating Systems Review* 13, 2 (1979), 20–25.

[10] Paulo Coelho, Tarcisio Ceolin Junior, Alysson Bessani, Fernando Dotti, and Fernando Pedone. 2018. Byzantine fault-tolerant atomic multicast. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 39–50.

[11] Paulo R Coelho, Nicolas Schiper, and Fernando Pedone. 2017. Fast atomic multicast. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE, 37–48.

[12] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.

[13] James Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 223–235. https://www.usenix.org/conference/atc12/technical-sessions/presentation/cowling

[14] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)* 36, 4 (2004), 372–421.

[15] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988), 288–323.

[16] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. 2021. Efficient Replication via Timestamp Stability. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. ACM, New York, NY, USA, 178–193.

[17] FastCast implementation [n. d.]. https://bitbucket.org/paulo_coelho/libmcast.

[18] Udo Fritzke and Philippe Ingels. 2001. Transactions on Partially Replicated Data based on Reliable and Atomic Multicasts. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*. 284–291.

[19] Udo Fritzke, Philippe Ingels, Achour Mostéfaoui, and Michel Raynal. 1998. Fault-tolerant total order multicast to asynchronous groups. In *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*. IEEE, 228–234.

[20] Alexey Gotsman, Anatole Lefort, and Gregory Chockler. 2019. White-Box Atomic Multicast. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 176–187.

[21] Rachid Guerraoui and André Schiper. 1997. Genuine Atomic Multicast. In *Proceedings of the 7th IEEE International Conference on Computer Communications and Networks*. IEEE, 840–847.

[22] Rachid Guerraoui and Andre Schiper. 1997. Total order multicast to multiple groups. In *Proceedings of 17th International Conference on Distributed Computing Systems*. IEEE, 578–585.

[23] Rachid Guerraoui and André Schiper. 2001. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science* 254, 1-2 (2001), 297–316.

[24] Vassos Hadzilacos and Sam Toueg. 1994. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Technical Report. Cornell University, Ithaca, NY, USA.

[25] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 245–256.

[26] Sandeep S Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical physical clocks. In *International Conference on Principles of Distributed Systems*. Springer, 17–32.

[27] Long Hoang Le, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Robert Soulé, and Fernando Pedone. 2019. Dynastar: Optimized dynamic partitioning for scalable state machine replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1453–1465.

[28] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *OSDI*. 467–483.

[29] Libevent library [n. d.]. https://libevent.org.

[30] Barbara Liskov and James Cowling. 2012. *Viewstamped replication revisited*. Technical Report. Technical Report MIT-CSAIL-TR-2012-021, MIT.

[31] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. 2012. Multi-ring paxos. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE, 1–12.

[32] Leandro Pacheco. 2023. *Scaling Strongly Consistent Replicated Systems*. Ph. D. Dissertation. Università della Svizzera italiana. https://sonar.ch/usi/documents/325574

[33] Leandro Pacheco, Raluca Halalai, Valerio Schiavoni, Fernando Pedone, Etienne Riviere, and Pascal Felber. 2016. GlobalFS: A Strongly Consistent Multi-site File System. In *Reliable Distributed Systems (SRDS), 2016 IEEE 35th Symposium on*. IEEE, 147–156.

[34] Fernando Pedone and André Schiper. 1999. Generic Broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99, formerly WDAG)*.

[35] PrimCast implementation [n. d.]. https://github.com/pacheco/primcast.

[36] Luis Rodrigues, Rachid Guerraoui, and André Schiper. 1998. Scalable atomic multicast. In *International Conference on Computer Communications and Networks*. 840–847.

[37] Nicolas Schiper and Fernando Pedone. 2007. Optimal atomic broadcast and multicast algorithms for wide area networks. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 384–385.

[38] Nicolas Schiper and Fernando Pedone. 2008. On the inherent cost of atomic broadcast and multicast in wide area networks. In *International conference on Distributed computing and networking (ICDCN)*. 147–157.

[39] Nicholas Schiper, Pierre Sutra, and Fernando Pedone. 2010. P-Store: Genuine Partial Replication in Wide Area Networks. In *Symposium on Reliable Distributed Systems (SRDS)*.

[40] Amazon Time Sync Service. [n. d.]. https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-the-amazon-time-sync-service/.

[41] Tokio asynchronous runtime [n. d.]. https://tokio.rs/.

[42] Robbert Van Renesse, Nicolas Schiper, and Fred B Schneider. 2014. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing* 12, 4 (2014), 472–484.

[43] White-Box implementation [n. d.]. https://github.com/imdea-software/atomic-multicast.