Journal of Internet Services and Applications, 2025, 16:1, doi: 10.5753/jisa.2025.5920

Generic Multicast: One Group Communication Primitive to Rule Them All

Jose Bolina [**Red Hat, Inc.** | ***jbolina@redhat.com***]
Douglas Antunes Rocha [**Independent Researcher** | ***douglasanr@ufu.br***]
Lasaro Camargos [**Weilliptic Inc.** | ***lasaro@weilliptic.com***]
Pierre Sutra [**Institut Polytechnique de Paris** | ***pierre.sutra@telecom-sudparis.eu***]

IBM BRASIL - Indústria, Máquina e Serviços Ltda., Rua Maria Luiza Santiago, 200 - 3o andar., Bairro Santa Lucia, Belo Horizonte, MG, 30360-740

Received: 15 April 2025 • **Accepted:** 10 October 2025 • **Published:** 04 December 2025

Abstract Group communication primitives have a central role in modern computing infrastructures. They offer a panel of reliability and ordering guarantees for messages, enabling the implementation of complex distributed interactions. In particular, atomic broadcast is a pivotal abstraction for implementing fault-tolerant distributed services. This primitive allows disseminating messages across the system in a total order. There are two group communication primitives closely related to atomic broadcast. Atomic multicast permits targeting a subset of participants, possibly stricter than the whole system. Generic broadcast leverages the semantics of messages to order them only where necessary, that is, when they do not commute at the application level (a conflict). In this paper, we propose to combine all these primitives into a single, more general one, called generic multicast. We formally specify the guarantees offered by generic multicast and present efficient algorithms. Compared to prior works, our solutions offer appealing properties in terms of time and space complexity. In particular, when a run is conflict-free, that is no two messages conflict, a message is delivered after at most three message delays. We explain the logic of of our algorithms, detail their main invariants, and prove them correct. We also present a variation that delivers messages across the system in an order consistent with real-time at the cost of a message delay. This variation is particularly interesting to implement partially-replicated data storage systems.

Keywords: Consensus, Multicast, Broadcast, Generalized Consensus.

1 Introduction

Atomic broadcast is a fundamental building block of modern computing infrastructures. This group communication primitive offers strong properties in the ordering and delivery of messages. Atomic broadcast finds usage in many storage systems, from file systems and relational databases to object stores and blockchains [Chandra *et al.*, 2007; Hunt *et al.*, 2010; Corbett *et al.*, 2013; Ongaro and Ousterhout, 2014; Camaioni *et al.*, 2024]. It ensures the scalability, high availability, and fault tolerance of these distributed systems. At its core, atomic broadcast guarantees the reliable delivery of messages in the same total order across the system. Such guarantees generalize into two natural and related group communication primitives.

The first primitive is called *atomic multicast*. Atomic multicast allows sending a message to a (possibly stricter) subset of the processes in the system. In this case, the ordering property becomes a partial order linking the pairs of messages having a joint destination (a common process). Atomic multicast helps to distribute the load and better leverage workload parallelism. It is used in geo-replicated and partially-replicated systems where data is stored in multiple partitions [Cowling and Liskov, 2012; Benz *et al.*, 2014].

The second common generalization of atomic broadcast is *generic broadcast* (aka., generalized consensus). This primitive is introduced in the work of Pedone and Schiper [2002] and Lamport [2005]. Here, the ordering property binds messages that conflict, that is, messages whose processing does not commute in the upper application layer. Generic broadcast leverages the semantics of messages to expedite delivery and improve overall performance [Moraru *et al.*, 2013].

Contributions In this work, we propose to combine the two primitives, atomic multicast and generic broadcast, into a new primitive called *generic multicast*. As atomic multicast, generic multicast permits sending a message to a subset of the processes in the system. Delivery is based on the semantics of messages, as in generic broadcast. This paper defines generic multicast and proposes efficient algorithmic solutions for message-passing systems.

With more details, we claim the following contributions:[1] *(i)* The definition of generic multicast for crash-prone distributed systems; *(ii)* A base solution that extends the timestamping approach of Skeen [Birman and Joseph, 1987]; *(iii)* Building upon this is an understandable, full-fledged, and fault-tolerant generic multicast algorithm; and *(iv)* An extended variation that delivers messages across the system in an order consistent with real-time.

Our algorithms follow well-established mechanisms proposed in the literature (*e.g.*, [Birman and Joseph, 1987; Fritzke *et al.*, 1998; Schiper and Pedone, 2007; Ahmed-Nacer *et al.*,

[1]This paper is an extended version of prior research [Bolina *et al.*, 2024].

		Latency				
	Algorithm	**conflict-free**	**collision-free**	**failure-free**	**generic**	**metadata per msg.**
failure-free	Skeen [Birman and Joseph, 1987]	5	3	5	×	$O(1)$
	Ahmed-Nacer *et al.* [2016]	3		5	√	$O(1)$
	Algorithm 1 (Section 4)	3		5	√	$O(1)$
fault-tolerant	FastCast [Coelho *et al.*, 2017]	8	4	8	×	$O(1)$
	White-Box [Gotsman *et al.*, 2019]	5+1	3+1	5+1	×	$O(1)$
	PrimCast [Pacheco *et al.*, 2023a]	5	3	5	×	$O(1)$
	Tempo [Enes *et al.*, 2021]	4	5	13	√	$O(m)$
	Algorithm 2 (Section 5)	3	5	11	√	$O(1)$

Table 1. Atomic multicast algorithms. (`+1`: additional delay to non-leader processes).

2016; Enes *et al.*, 2021]). This makes them simple and understandable. Moreover, they are thrifty in the amount of metadata they consume to order messages. The base (non-fault-tolerant) solution delivers a message in three message delays when no two messages conflict. We show that tolerating failures is possible without incurring additional communication steps. To the best of our knowledge, this is the first time such a low latency is attained for conflict-free scenarios (see Table 1 for a comparison with prior works). Further, we explain how to deliver messages in an order consistent with real-time at the cost of a message delay. Such a variation ensures that when a message is multicast after another gets delivered, no process may deliver them in the converse order. As we illustrate in the paper, such a property is key to implement linearizable partially-replicated storage systems.

Outline The remainder of this work is organized as follows. Section 2 reviews existing literature on atomic multicast. Section 3 presents the system model and defines generic multicast as well as some related properties of interest. Section 4 depicts our base (non-fault-tolerant) solution. Section 5 extends the base solution into a full-fledged fault-tolerant algorithm. Additionally, this section argues about the performance of the algorithm and its correctness proofs. Section 6 explains how messages can be delivered in an order consistent with real-time. We close in Section 7 with a summary of the paper and a prospect of future works.

2 Background

The literature on atomic multicast is rich, finding its roots in the early works on group communication primitives. In what follows, we offer a brief tour of the topic, underlining the key ideas and algorithmic principles. Table 1 lists the most recent solutions and compares them against the two algorithms we cover in Sections 4 and 5. The first part of Table 1 lists non-fault-tolerant algorithms, and the second part lists the fault-tolerant versions. Figure 1 gives the genealogy of the two algorithms.

Early Solutions The first solution to atomic multicast appears in the work of Birman and Joseph [1987]. In this work, the authors describe an unpublished algorithm by Dale Skeen that uses priorities to order the delivery of messages. The algorithm employs a two-phase approach. In the first phase, the sender disseminates the message to its destination group and awaits their responses. Upon receiving such a message, an algorithm assigns it a priority greater than prior messages and sends this priority back to the sender. The sender collects responses from the destination groups. It computes the highest priority among all the proposals and informs the destination group. Each process in the destination group assigns this new priority to the message. In a process, messages are delivered in the order of priority, from lowest to highest. This algorithm can take up to five message delays to deliver a message.

In modern terms, the solution of Skeen is a *timestamping* algorithm. This early solution does not tolerate failures: when the sender fails, the algorithm stalls. Birman and Joseph [1987] propose that another process takes over the role of the sender when it fails. This requires a synchronous system. However, in general, distributed systems are partially synchronous. In this context, one cannot accurately detect if a process in the destination group has failed. To deal with it, Guerraoui and Schiper [1997] proposes to partition the recipients of a message into (disjoint) consensus groups. Each group maintains a clock that it uses and advances to propose timestamps. A similar solution is proposed by Fritzke *et al.* [1998] and Delporte-Gallet and Fauconnier [2000]. From a high-level perspective, all these approaches can be seen as replicating the logic of a Skeen process over a reliable group of machines.

Message Semantics In Skeen's approach, each process maintains a logical clock to assign timestamps. Once the final timestamp of a message is known, the logical clock is bumped to a higher value. This ensures progress as, otherwise, the message would stall until enough (prior) messages get timestamped. In this schema, right before the clock is bumped, another message can sneak in and retrieve an earlier timestamp, which delays delivery. This situation is due to contention, creating a convoy effect in the system [Ahmed-Nacer *et al.*, 2016]. To deal with it, Ahmed-Nacer *et al.* [2016] proposes to leverage the semantics of messages. If, from the application's perspective, the two messages commute, they do not need to wait for each other. Such an idea finds its root in the generalizations of atomic broadcast [Pedone and Schiper, 1999; Lamport, 2005]. These algorithms use the semantics of messages to deliver them earlier. Building upon this, Ahmed-Nacer *et al.* [2016] defines generic multicast as a variation of atomic multicast that understands the message semantics. However, the definition in [Ahmed-Nacer *et al.*, 2016] only applies to the failure-free case. We extend it to failure-prone systems in Section 3. Notice that the mention of a generalized version of atomic multicast can be traced back to earlier works (*e.g.*, [Schiper, 2009]).

Recent Progress Guerraoui and Schiper [1997] execute a second agreement per consensus group to bump the clock. Schiper and Pedone [2007] observe that this is not always necessary. In fact, when a message is addressed to a single consensus group, no timestamping mechanism is needed. Coelho *et al.* [2017] propose the FastCast algorithm. This solution delivers messages in fewer communication steps by leveraging a speculative path to update the logical clock. The fast path executes concurrently with a slower (conservative) path. The message is delivered in four message delays when the fast path correctly guesses the final timestamp. An improvement is made to the above schema by Gotsman *et al.* [2019]. This algorithm opens the consensus "black box" to make additional optimizations. In the absence of collisions, the solution in [Gotsman *et al.*, 2019] delivers a message in just three message delays at the leader of a consensus group. Non-leader members need an additional communication step. PrimCast [Pacheco *et al.*, 2023a] is the best collision-free solution known to date. It cuts the additional message delay by exchanging commit acknowledgment messages across consensus groups (and not just at the leaders as in [Gotsman *et al.*, 2019]). The authors also propose a technique of loosely synchronized logical clocks to reduce the convoy effect.

When multiple messages are addressed concurrently to a process, the above algorithms are slower ("failure-free" column in Table 1). To avoid this, Tempo [Enes *et al.*, 2021] leverages the semantics of messages. Messages can access one or more data items, among m (see Table 1). Two messages conflict, *i.e.*, do not commute, if and only if they access a common item. When messages collide but do not conflict, Tempo can deliver them in four message delays ("collision-free" column in Table 1).

The work in [Sutra, 2022] characterizes the minimal synchrony assumptions to solve atomic multicast. In particular, this work shows that, in some cases, weaker assumptions than the standard partitioning into consensus groups are possible. The present paper does not investigate such systems.

3 Generic Multicast

This work introduces the generic multicast problem in failure-prone systems. Generic multicast is a flexible group communication primitive that takes the best of two worlds. Like atomic multicast, it permits addressing a message to a subset of the processes in the system. Additionally, the semantics of the messages are taken into account, as in generic broadcast, to order them only when necessary.

This section presents the system model and defines the generic multicast problem. It then illustrates this primitive in the context of distributed storage. Further, we discuss how to implement generic multicast efficiently.

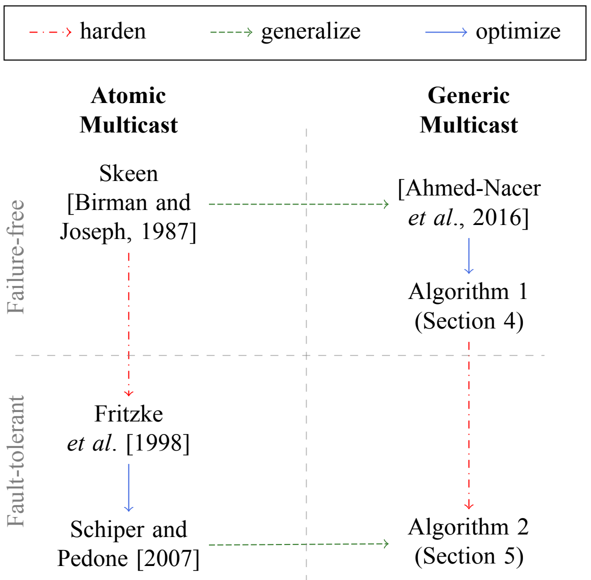

Figure 1. Structured derivations of generic multicast.

3.1 System Model

The set $\Pi = \{p_1, p_2, \dots, p_{n\geq 2}\}$ contains all the processes in the distributed system. A process may fail by crashing, halting its execution. If this happens, the process is *faulty*. Otherwise, it is *correct*. Processes do not share a common memory and solely rely on passing messages through communication channels to exchange information. Any two processes are connected with a channel. A call to $Send\langle m, p\rangle$ sends the message m to the process p. The event $Receive\langle m, p\rangle$ is triggered when message m is received from p. Channels are *reliable*. This means that if a correct process p sends a message to a correct process q, then eventually q receives this message.

The system is partially synchronous. Partial synchrony is modeled with the help of failure detectors [Chandra and Toueg, 1996]. For pedagogical purposes, we first consider a failure-free system in which all processes are correct (Section 4). Later, we augment the system with failure detectors and quorums (the details are in Section 5).

3.2 Definition

Generic multicast permits the dissemination of a message across the system with strong liveness and safety guarantees. In what follows, we formally define the communication primitive and illustrate its usage in the context of a storage system.

We note $\mathcal{M}$ the set of messages disseminated with generic multicast. Each such message m carries its source ($m.src \in \Pi$), a unique identifier ($m.id$), as well as the *destination group* ($m.d \subseteq \Pi$). There are two operations at the interface: GM-Send(m) multicasts some message m to its destination group, that is, the processes in $m.d$. GM-Deliver(m) triggers at a process when message m is delivered locally. As is common, we consider that processes multicast different messages and that a message is multicast at most once.

In generic multicast, delivery is parameterized with a conflict relation ($\sim$) that captures the semantics of messages from an application perspective [Pedone and Schiper, 1999; Lamport, 2005]. Relation $\sim$ is irreflexive and symmetric over $\mathcal{M}$. It is such that any two conflicting messages must have a recipient in common: $m \sim m' \implies m.d \cap m'.d \neq \emptyset$.

For a process p and two conflicting messages $m \sim m'$, with $p \in m.d \cap m'.d$, we note $m \rightarrow_p m'$ when p delivers message m before it delivers message m'. This relation tracks the *local delivery order* at process p. Notice that $m \rightarrow_p m'$ holds even if p never delivers message m'. The *global delivery order* captures how messages are delivered across the system. This relation is defined as: $\rightarrow = \bigcup_{p \in \Pi} \rightarrow_p$.

Generic multicast ensures the following properties on the dissemination of messages:

- **Integrity**: For some message m, every process $p \in m.d$ invokes GM-Deliver(m) at most once and only if some process invoked GM-Send(m) before.
- **Termination**: If a correct process p GM-Send(m) or GM-Deliver(m) some message m, eventually every correct process $q \in m.d$ executes GM-Deliver(m).
- **Ordering**: Relation $\rightarrow$ is acyclic.

Generic multicast establishes a partial order among delivered messages. In simple terms, it ensures that conflicting messages are delivered in a total order, while non-conflicting ones are delivered in any order. In both cases, a message is delivered reliably to all the processes in the destination group.

Flexibility Generic multicast is a flexible communication primitive. If the conflict relation binds any two messages having a joint destination ($m \sim m' \iff m.d \cap m'.d \neq \emptyset$), we obtain atomic multicast. Conversely, if there is no conflict ($\sim = \emptyset$), the above definition is the one of reliable multicast. These observations also extend naturally to the case where messages are addressed to everybody in the system. In such a case, the specification boils down to the one of generic broadcast [Pedone and Schiper, 1999]. Then, if all messages conflict, we obtain atomic broadcast, and when there are no conflicts, the specification of reliable broadcast.

From what precedes, by adjusting the conflict relation and/or the destination groups, we can adapt the behavior of generic multicast to suit an application's needs. This permits tailoring an implementation to a specific context without modifying it. In the following section, we illustrate such an idea in the case of a storage system.

3.3 Usage: A Key-Value Store

To illustrate the above definitions, let us consider a key-value store. Such a service is common in cloud infrastructures where it can be accessed concurrently by multiple remote clients (typically, application backends). Examples of key-value stores include AWS S3 [Amazon, 2008], etcd [The etcd Authors, 2014], and Apache Cassandra [Lakshman and Malik, 2010].

A key-value store maps a set of keys to a set of values. In detail, its interface consists of three operations: a call to `read` (k) reads the content stored under key k, operation `write` (k,v) maps key k to value v, and `cas` (k,u,v) executes a compare-and-swap, that is it stores v under key k, provided the associated value was previously u. Additionally, the interface includes a batch operator `start`...`end` which permits grouping several of these operations.

Modern key-value stores replicate data across several availability zones and/or geographical regions. This improves fault tolerance and data locality. A standard approach to implementing such systems is to rely on the use of atomic broadcast in conjunction with atomic commitment. For instance, this is the design of Spanner [Corbett *et al.*, 2013]. In Spanner, each unit of replication, or *tablet*, is replicated across multiple data centers. Operations on a tablet are ordered with the Paxos consensus algorithm. When a batch of operations executes over multiple tablets, the replicas run two-phase commit (2PC) [Gray, 1978; Lampson and Sturgis, 1979] to agree on committing or aborting its changes.

Atomic multicast provides an alternative design [Cowling and Liskov, 2012; Benz *et al.*, 2014]. Upon executing an operation, a message is multicast to the replicas holding a copy of the corresponding data items. When the message is delivered, replicas apply the operation locally. (If the message contains a batch, an additional phase is needed, as detailed in Section 6.) Thanks to the ordering property of atomic multicast, this approach also provides strong consistency to the service clients.

In this context, generic multicast can be used as a drop-in replacement to atomic multicast. For this, we need to define the conflict relation ($\sim$) appropriately. Given an operation c, $\texttt{isRead}(c)$ evaluates to true when c is a read operation. We write $\texttt{key}(c)$ the key accessed by c. Operations c and d conflict when they access the same key and one of them is an update (a write or a compare-and-swap). Formally,

$$c \sim d \triangleq \begin{array}{l} \wedge\ \texttt{key}(c) = \texttt{key}(d) \\ \wedge\ \neg\ (\texttt{isRead}(c) \wedge \texttt{isRead}(d)) \end{array} \tag{1}$$

For some message m, we write $\texttt{ops}(m)$ the operations m disseminates. Given two distinct messages m and m', the conflict relation is then defined as:

$$m \sim m' \triangleq \exists c, d \in \texttt{ops}(m) \times \texttt{ops}(m') : c \sim d \tag{2}$$

With generic multicast, operations on the key-value store that do not access the same items commute. Hence, this group communication primitive permits ordering messages only where needed; this is more flexible and potentially also faster. If the workload is contended on some specific keys, the system coordinates only the access to those hot items. If later, the workload becomes uniform, there is no need to revisit the architecture, as the same guarantees still hold.

In the remainder of the paper, we explain how to implement generic multicast efficiently. Before detailing these solutions, we first discuss the properties of interest they should have.

3.4 Implementation Properties

Atomic multicast satisfies all the requirements of generic multicast. Unfortunately, it also orders messages more than necessary. To avoid this, Pedone and Schiper [1999] introduces a property that captures when an implementation permits non-conflicting messages to be delivered in different orders. The definition is given for generic broadcast. It extends naturally to our context as follows:[2]

> (*Rigidness*) Consider two processes p and q and two non-conflicting messages m and m' with $\{p, q\} \subseteq m.d \cap m'.d$. There exist a run r in which p delivers m before m' while q delivers m' before m.

Another property of interest is with respect to the destination group. Indeed, it is possible to implement generic multicast by delivering messages first everywhere, with atomic (or generic) broadcast, and then filtering them out based on their destinations. However, this defeats the purpose of targeting a subset of the system. In [Guerraoui and Schiper, 2001], the authors introduce a minimality property that rules out such approaches.

> (*Minimality*) In every run r, if some correct process p sends or receives a (non-null) message in r, there exists a message m multicast in r with $p \in m.d \cup \{m.src\}$.

The two properties above are of interest from an implementation perspective. They ensure the communication primitive is flexible enough and appropriately leverages the semantics of messages it disseminates. In what follows, we complement these with metrics that capture performance.

3.5 Time Complexity

In [Gotsman *et al.*, 2019], the authors define the notions of failure-free and collision-free latency. In what follows, we extend these metrics to the context of generic multicast.

The two metrics are defined when the system is stable, *i.e.*, when there are no failures and both the processes and the network behave synchronously. Runs are classified whether they contain concurrent and/or conflicting messages or not.

The *delivery latency* of a message is the time interval between the moment it is multicast and delivered everywhere.[3] The *conflict-free latency* is the highest delivery latency for a message when there are no conflicting messages. A message m precedes a message m' if m is last delivered before m' is multicast. [4] Two messages m and m' are concurrent when neither m precedes m' nor the converse holds. The *collision-free latency* is the highest delivery latency for a message when there is no concurrent message with a common destination. Last, the *failure-free latency* is the highest delivery latency for a message in the presence of concurrent and conflicting messages.

To illustrate the above notions, let us go back to our storage example (in Section 3.3). The failure-free latency defines an upper bound on the time an operation takes to return in the common case. If the operation does not encounter a concurrent operation when accessing the same replicas, it may return earlier; this is the collision-free latency. Now, in case the operation accesses a cold item, its response can be even faster: in this situation, data is identical everywhere, which is similar to running the operation solo from the initial state. This optimal case is measured with the conflict-free latency.

Lower bounds As mentioned earlier, atomic broadcast trivially reduces to generic multicast. Hence, any known lower bound on this communication primitive applies to generic multicast. It is well-known that generic broadcast can deliver, at best, a message after a single round-trip, that is, two message delays, in a conflict-free (or collision-free) scenario [Lamport, 2005]. In all the other cases, at least three message delays are necessary [Lamport, 2006].

4 A Base Solution

This section introduces a base solution to generic multicast. This algorithm follows the standard schema invented by Skeen [Birman and Joseph, 1987]: Given a message, each process in its destination group proposes a timestamp. The highest such proposal is the final timestamp of the message. Message delivery happens in the order of their final timestamps.

In Skeen's schema, the clock ticks every time a message is received. Our key observation is that generic multicast only needs this to happen when a conflict occurs. Moreover, conflict-free messages are delivered in parallel without waiting—in contrast to Skeen's solution. Both mechanisms help to reduce latency and they diminish the convoy effect in the communication primitive.

Algorithm 1 depicts an implementation of generic multicast. This algorithm works when processes are all correct. Below, we present its variables, the overall logic, and then we detail the internals.

Overview At a process, Algorithm 1 makes use of the following four local variables.

- K: A logical clock to propose a timestamp for each message.
- $Pending$: The messages that do not have a timestamp assigned to them so far.
- $Delivering$: The messages whose timestamp is decided and are ready for delivery.
- $PreviousMsgs$: The messages received since a conflict was detected. By construction, this set only contains messages that do not conflict with each other.

Using the above variables, Algorithm 1 proposes and decides timestamps for the messages submitted to generic multicast. In a nutshell, each newly submitted message is first disseminated to its destinations. These processes advance their clocks to propose a timestamp for the message. Such proposals are then gathered, and the highest one defines the final timestamp. A message is then delivered in the order of its timestamp.

As mentioned earlier, the logical clock advances only when a conflict is detected. Namely, when a conflict is detected, the set $PreviousMsgs$ is cleared, and the clock advances. This mechanism reduces the convoy effect in the communication primitive. We further detail this next.

[2] The property is called "strictness" in [Pedone and Schiper, 1999].

[3] In [Gotsman *et al.*, 2019], the authors consider the moment the message is first delivered. Here, we follow the definition proposed in [Pacheco *et al.*, 2023a].

[4] The term "before" refers here to real-time ordering.

Algorithm 1 Base generic multicast – code at p.

```
1: Variables:
2: $K \leftarrow 0$
3: $Pending \leftarrow \emptyset$
4: $Delivering \leftarrow \emptyset$
5: $PreviousMsgs \leftarrow \emptyset$

6: procedure GM-Send(m)
7:     for all $q \in m.d$ do
8:         $Send\langle \texttt{Begin}(m), p\rangle$

9: when $Receive\langle \textsc{Begin}(m), q\rangle$
10:    if $\exists\, m' \in PreviousMsgs : m \sim m'$ then
11:        $K \leftarrow K + 1$
12:        $PreviousMsgs \leftarrow \emptyset$
13:    $PreviousMsgs \leftarrow PreviousMsgs \cup \{m\}$
14:    $Pending \leftarrow Pending \cup \{(m, K)\}$
15:    $Send\langle \texttt{Propose}(m, K), q\rangle$

16: when $\forall q \in m.d : Receive\langle \textsc{Propose}(m, ts), q\rangle$
17:    $ts_f \leftarrow \max(\{ts \mid Receive\langle \texttt{Propose}(m, ts), q\rangle\})$
18:    for all $q \in m.d$ do
19:        $Send\langle \texttt{Deliver}(m, ts_f), q\rangle$

20: when $Receive\langle \textsc{Deliver}(m, ts_f), p\rangle$
    pre: $(m, \_) \in Pending$
21:    if $ts_f > K$ then
22:        if $\exists m' \in PreviousMsgs : m \sim m'$ then
23:            $K \leftarrow ts_f + 1$
24:            $PreviousMsgs \leftarrow \emptyset$
25:        else
26:            $K \leftarrow ts_f$
27:    $Pending \leftarrow Pending \setminus \{(m, \_)\}$
28:    $Delivering \leftarrow Delivering \cup \{(m, ts_f)\}$

29: procedure GM-Deliver(m)
    pre: $\wedge\ \exists t : (m, t) \in Delivering$
         $\wedge\ \forall m', t' : (m', t') \in Delivering \cup Pending$
                 $\implies (m \not\sim m' \vee (m, t) < (m', t'))$
30:    $Delivering \leftarrow Delivering \setminus \{(m, t)\}$
```

Internals To disseminate some message m, a process p invokes at line 6 operation GM-Send(m). The operation sends a message Begin(m) to all the processes in $m.d$ using the underlying $Send$ communication primitive. When this happens, we say that p is the *sender* of message m.

At line 9, the handler triggers when process p receives a Begin(m) message from m's sender, q. In such a case, p first checks if m conflicts with a previously received message. This computation is at line 10, using the $PreviousMsgs$ variable. If the process identifies a conflict, it increments its clock and clears the $PreviousMsgs$ set in lines 11 to 12. Then, the process stores the message m and its timestamp proposal, which is the value of its clock. The process sends to the sender q the proposal for m at line 15 with a Propose message.

When the sender gathers a proposal from all the processes in $m.d$, it computes the final timestamp of m by selecting the highest proposed timestamp. The sender then disseminates the decision to all the processes with a Deliver message. Such a computation is in lines 16 to 19.

Upon receiving the final timestamp ts_f for message m, a process checks if its clock is lower than ts_f. If this happens to be the case, the clock is advanced to ts_f. In case a conflict is detected, the clock is also incremented by one, and the $PreviousMsgs$ set is cleared. This step ensures that no earlier message is missed: when the final timestamp is known, all the messages before m and conflicting with it are stored locally.

Once the above steps are executed, the message is ready to get delivered at line 29. The precondition in this line ensures that delivery happens in the order of the final timestamps. In case two timestamps are tied between conflicting messages, the identifier of each message provides a deterministic order. This means that $(m, t) < (m', t')$ reads as $t < t' \vee (t = t' \wedge m.id < m'.id)$.

Discussion Variable $PreviousMsgs$ can grow arbitrarily large when there is no conflict. This may negatively impact memory usage. In practice, it suffices to bump the clock and clear the set periodically. Because processes cannot unilaterally bump their clocks, such a garbage-collection mechanism ought to be deterministic: it can happen at a given interval or by broadcasting an appropriate command across the system.

Algorithm 1 does not restrict the conflict relation. Therefore, it is possible to build the storage example (Section 3.3) utilizing the conflict relation of Equation (2).

For Algorithm 1, we do not present correctness arguments. They are detailed in the next section, where we present a complete fault-tolerant solution.

5 Dealing With Failures

We now present a fault-tolerant algorithm to solve the generic multicast problem. This solution merges the logic of Algorithm 1 with some of the ideas introduced in earlier works (*e.g.*, [Fritzke *et al.*, 1998; Schiper and Pedone, 2007]).

In what follows, we first detail the necessary additional assumptions on the system model to deal with process failures. Then, we introduce our solution, argue about its correctness, and prove that its performance matches the results in Table 1.

5.1 Additional Assumptions

Fault-tolerance To be fault-tolerant, we need to strengthen our computational model. A standard assumption [Défago *et al.*, 2004] is to assume that the system is partitionable into groups. Internally, each such group is robust enough to solve consensus. The destination of a message is then defined as the union of one (or more) of these consensus groups.

In detail, we assume a partition $\Gamma = \{g_1, \ldots, g_m\}$ into groups of Π ensuring that:

- (*non-empty*) $\forall g \in \Gamma : g \neq \emptyset$
- (*complete*) $\Pi = \bigcup_{g \in \Gamma} g$
- (*disjoint*) $\forall g_i, g_j \in \Gamma : i \neq j \implies g_i \cap g_j = \emptyset$
- (*covering*) $\forall m \in \mathcal{M} : \exists \mathcal{G} \subseteq \Gamma : m.d = \cup_{g \in \mathcal{G}}\, g$
- (*reliable*) $\forall g \in \Gamma : \wedge$ consensus is solvable in g
  $\wedge$ a process in g is correct

In light of the above assumptions, generic broadcast is solvable in every group $g \in \Gamma$. Hereafter, we assume that such an instance exists per group. For some group g, we note GB-Send$_g(m)$ and GB-Deliver$_g(m)$ respectively the operation to broadcast and deliver a message m using generic broadcast in group g.

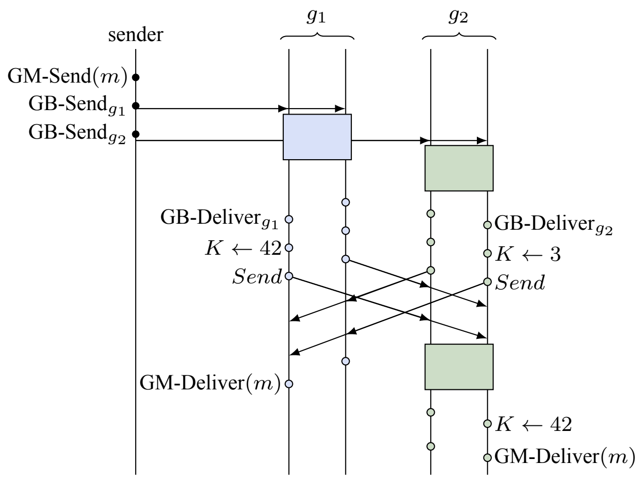

Figure 2. A run of Algorithm 2.

Conflict relation In what follows, we consider that messages convey a single operation and that they all access the same key. That is, the conflict relation ($\sim$) abides by Equations (1-2), with $|\mathtt{ops}(m)| = 1$, and there is a single key. The other cases are covered in Section 5.5.

5.2 Algorithm

Our solution is presented in Algorithm 2. Below, we provide an overview of the protocol, detail its internals, and later argue about its correctness.

Overview Algorithm 2 uses the variables listed in lines 1 to 4. Some of them have the same usage as in Algorithm 1.

- K: A logical clock.
- $PreviousMsgs$: The messages received since a conflict was detected.
- Mem: This set stores information to process each message. It plays the joint roles of $Pending$ and $Delivering$ sets of Algorithm 1.

Algorithm 2 follows the standard transformation invented in [Guerraoui and Schiper, 1997]. Such a transformation makes Skeen's approach fault-tolerant by considering subsets of the destination group that are large enough to agree on a timestamp proposal. This is the purpose of the assumptions made in Section 5.1.

In detail, for each destination group $m.d$, there exists a partition into a set of (consensus) groups $\mathcal{G}$ such that $m.d = \cup_{g \in \mathcal{G}}\, g$. In each group g, processes agree on a timestamp proposal for message m. Agreement takes place using generic broadcast in group g. Then, as in Algorithm 1, the final timestamp ts_f of message m is the highest of such proposals. To deliver the message m, it is necessary to know all the messages with a lower final timestamp than m. For this, each group g bumps its clock using generic broadcast a second time. In case a group has a clock high enough, it simply does nothing.

Figure 2 illustrates the above logic. In this failure-free scenario, two consensus groups, g_1 and g_2, partition the destination group of message m. Each group computes a timestamp proposal using generic broadcast: group g_1 proposes 42, while g_2 suggests 3. This computation occurs for g_1 in

Algorithm 2 Fault-tolerant generic multicast – code at p

```
 1: Variables:
 2: $K \leftarrow 0$
 3: $Mem \leftarrow \emptyset$
 4: $PreviousMsgs \leftarrow \emptyset$

 5: procedure GM-Send(m)
 6:     let $\mathcal{G} \subseteq \Gamma$ such that $m.d = \cup_{g \in \mathcal{G}}\, g$        $\triangleright$ $\mathcal{G}$ is unique
 7:     for all $g \in \mathcal{G}$ do
 8:         GB-Send$_g$($\mathtt{Begin}(m)$)

 9: when GB-Deliver$_g$($\mathtt{Begin}$ (m))
10:     if $\exists\, m' \in PreviousMsgs : m \sim m'$ then
11:         $K \leftarrow K + 1$
12:         $PreviousMsgs \leftarrow \emptyset$
13:     $PreviousMsgs \leftarrow PreviousMsgs \cup \{m\}$
14:     $Mem \leftarrow Mem \cup \mathtt{Propose}(m, K)$
15:     for all $q \in m.d$ do
16:         $Send\langle \mathtt{Propose}(m, K), q\rangle$

17: when $Receive\langle \textsc{Propose}(m, \_), \_\rangle$
    pre: $\wedge\ \mathtt{Propose}(m, \_) \in Mem$
         $\wedge\ m.d = \cup \{g \in \Gamma \mid \exists q \in g : Receive\langle \mathtt{Propose}(m, \_), q\rangle\}$
18:     $ts_f \leftarrow max(\{t \mid Receive\langle \mathtt{Propose}(m, t)\rangle\})$
19:     $Mem \leftarrow Mem \cup \mathtt{Deliver}(m, ts_f) \setminus \mathtt{Propose}(m, \_)$
20:     if $K < ts_f$ then
21:         let $g \in \Gamma$ such that $p \in g$        $\triangleright$ $g$ is unique
22:         GB-Send$_g$($\mathtt{Advance}(ts_f)$)

23: when GB-Deliver$_g$($\textsc{Advance}(t)$)
24:     if $t > K$ then
25:         $K \leftarrow t$
26:         $PreviousMsgs \leftarrow \emptyset$

27: procedure GM-Deliver(m)
    pre: $\wedge\ \exists t \leq K : \mathtt{Deliver}(m, t) \in Mem$
         $\wedge\ \forall m', t' : \mathtt{Propose}|\mathtt{Deliver}(m', t') \in Mem$
              $\implies (m \not\sim m' \vee (m, t) < (m', t'))$
28:     $Mem \leftarrow Mem \setminus \mathtt{Deliver}(m, t)$
```

the blue box. In parallel, g_2 also reaches an agreement on the proposal using the green box. Then, the two consensus groups exchange their proposal and set the final timestamp of m to 42. At group g_2, the clock is lower than 42. Thus, g_1 can immediately deliver the message. Group g_2 has initially proposed 3. As a consequence, it needs to bump its clock using generic broadcast a second time (green box at the bottom of Figure 2).

Internals A message m starts its journey once it is multicast at line 5. When this happens, the sender p first computes the partition $\mathcal{G}$ of the destination group $m.d$. Then, for each consensus group $g \in \mathcal{G}$, p broadcasts a $\mathtt{Begin}(m)$ message to g. This disseminates message m to its destination group to compute timestamp proposals. If p fails in the loop at line 7, a recovery is needed. We will explain the recovery procedure later in this section.

Upon receiving a $\mathtt{Begin}(m)$ message, a process p looks for conflicts in the $PreviousMsgs$ set (line 10). If a conflict exists, p increments its clock and clears $PreviousMsgs$ (lines 11 and 12). Then, m is added to $PreviousMsgs$ (line 13). These steps are similar to the ones taken in Algorithm 1, and they are needed to ensure that conflicting messages end up with different final timestamps.

The timestamp proposal for m is set to the value of the clock K. This proposal is then stored at the local process p in variable Mem (line 14). One can show that the processes in consensus group g compute the exact same proposal for m. As a consequence, there is no need to disseminate the

Algorithm 3 Recovery mechanism

```
1: procedure recover($m$)
     pre: $\land \exists\, m : \texttt{Propose}(m, \_) \in Mem$
          $\land\ m.src \in \mathcal{D}$
2:   for all $g \in \Gamma : g \subseteq m.d$ do
3:     if $\nexists p \in g : Receive\langle \texttt{Propose}(m, \_), p\rangle$ then
4:       $GB\text{-}Send_g(\texttt{Begin}(m))$
```

group's proposal internally. In lines 15 to 16, p sends the proposal to all the other processes in the destination group. It also sends the proposal to itself to move the message to the decision phase.

Deciding the final timestamp for message m happens in the handler at lines 17 to 22. The handler has a precondition that requires that a timestamp proposal is known for each consensus group in the partition of the destination group. The final timestamp of m is stored in variable ts_f and set to the highest such proposal (line 18). If some consensus group g is late, that is, its clock is lower than ts_f, it must bump its clock. This ensures that all the messages lower than ts_f are known locally when m is ready to be delivered (line 27). Clock synchronization happens in lines 20 to 26. It broadcasts an $\texttt{Advance}(ts_f)$ message within the local group.

Within a consensus group g, clocks are synchronized using generic broadcast. This permits processes to make the same timestamp proposal for a message addressed to the group. More formally, if p and q in g GB-Deliver$_g$ the same messages, then variable K is in the same state. For this, we need to carefully define conflicts in the generic broadcast primitive to ensure the group behaves as a unit. Equation (3) specifies how relation ($\sim_{gb}$) is set to satisfy the requirements for two (distinct) messages m and m'.

$$m \sim_{gb} m' \triangleq \begin{aligned}[t] &\lor m = \texttt{Advance}(\_) \\ &\lor m' = \texttt{Advance}(\_) \\ &\lor \land\, m = \texttt{Begin}(x) \\ &\quad \land\, m' = \texttt{Begin}(y) \\ &\quad \land\, x \sim y \end{aligned} \tag{3}$$

Message m ends its journey when the handler at line 27 triggers. This happens when the clock is higher than m's final timestamp. As in Algorithm 1, such a precondition ensures that every message with a lower final timestamp is known locally. (This precondition can be added to Algorithm 1, but it is vacuously true.) The rest of the handler is identical to the failure-free case; message delivery occurs in the timestamp order.

5.3 Failure Recovery

Algorithm 2 is always safe. However, the algorithm would block when a process fails without a recovery procedure. In detail, if the sender crashes before it finishes the loop at line 7, some processes in the destination group may never compute a timestamp proposal. To avoid this, we need a recovery mechanism. This mechanism is detailed in Algorithm 3.

Algorithm 3 makes use of an unreliable failure detector, denoted $\mathcal{D}$. This failure detector returns a list of processes that are suspected to have failed [Chandra and Toueg, 1996]. Failure detection $\mathcal{D}$ must ensure that a faulty process cannot remain unsuspected forever. Notice that this does not require any assumption on the underlying system. In particular, returning Π is a valid implementation of failure detector $\mathcal{D}$. Better failure detection is interesting for performance, but does not impact correctness or liveness.

At a process p, the recovery mechanism in Algorithm 3 works as follows: It triggers when for some message m, m is pending ($\texttt{Propose}(m, \_) \in Mem$) and its sender is suspected ($m.src \in \mathcal{D}$). In such a case, for every group g partitioning the destination, if p did not receive a timestamp proposal from g, p broadcasts a $\texttt{Begin}(m)$ message to g (lines 2 to 3). When process p is correct, the missing groups deliver a $\texttt{Begin}(m)$ message at line 9 and then send their timestamp proposals. Otherwise, recovery is attempted again by another process, until it succeeds (which happens eventually from the assumptions in Section 5.1).

5.4 Correctness

In what follows, we present the correctness proofs of Algorithm 2. Algorithm 2 was also verified using TLA$^+$. The specification can be found online[5].

5.4.1 Safety

We start the proof of Algorithm 2 with some invariants:

Base 1 At a process p, relation $\rightarrow_p$ is irreflexive.
Base 2 The logical clock at a process never decreases.
Base 3 If a process send messages $\texttt{Propose}(m, t)$ and $\texttt{Propose}(m', t')$ with $m \sim m'$, then $t \neq t'$.
Base 4 Consider a consensus group g. If $p \in g$ sends message $\texttt{Propose}(m, t)$ and $q \in g$ also sends $\texttt{Propose}(m, t')$, then $t' = t$.

SAFa Processes in the destination group of a message m agree on the final timestamp of m.
SAFb A process delivers a message only if it has already delivered all the conflicting messages with a lower timestamp.

SAF For any two conflicting messages m and m' delivered in a run with respectively timestamps t and t', if $m \rightarrow_p m'$ then $(m, t) < (m', t')$.

Proof Invariant Base 1. When p delivers some message m, $\texttt{Deliver}(m, t)$ must be stored in Mem (line 27). Moreover, when this takes place, $\texttt{Deliver}(m, t)$ is removed from Mem (line 28). $\texttt{Deliver}(m, t)$ is added at line 19. This requires $\texttt{Begin}(m)$ in Mem. When p executes line 19, $\texttt{Begin}(m)$ is removed from Mem. Such a message is added when the block in lines 9 to 16 executes. This block is triggered when $\texttt{Begin}(m)$ is delivered by Generic Broadcast in the (consensus) group of process p. Because Generic Broadcast delivers each message at most once, this happens at most once. From what precedes, the delivery of m occurs at most once, implying that $\rightarrow_p$ is irreflexive at p. ☐

Proof Invariant Base 2. Follows from the pseudo-code of Algorithm 2. The clock (variable K) is either incremented by one (line 11), or bumped to a higher value (lines 24 to 25). ☐

[5]https://github.com/jabolina/mcast-tlaplus

Proof Invariant Base 3. Consider that process p sends messages $\texttt{Propose}(m,t)$ and $\texttt{Propose}(m',t')$ during a run, with $m \sim m'$. Such messages are disseminated at line 16, after that $\texttt{Begin}(m)$ and $\texttt{Begin}(m')$ were delivered at line 9 by Generic Broadcast. A message is delivered by Generic Broadcast at most once. Without lack of generality, consider that p executes line 9 for m before m'. Let τ and $\tau' > \tau$ be the respective points in time at which this takes place.

Recall that each block executes in full atomically, and thus they are not interleaved. At time τ, process p sends a $\texttt{Propose}(m,t)$ message at line 16. Note K_τ the value of the clock variable K on process p at time τ. According to the block in lines 9 to 16, $K_\tau = t$.

By Invariant Base 2, variable K never decreases. Note $K_{\tau'-1}$ the value of the clock variable on process p right before it executes the delivery of $\texttt{Begin}(m')$ at line 9. There are two cases to consider. (case $K_{\tau'-1} > K_\tau$.) Using the same reasoning as above, we have $t' \geq K_{\tau'-1}$. (case $K_{\tau'-1} = K_\tau$.) Because $m' \sim m$, process p must execute lines 11 to 12 before sending $\texttt{Propose}(m',t')$. It follows that $t' = K_{\tau'-1} + 1$, as required. □

Proof Invariant Base 4. The intuition for this invariant is as follows: Due to Equation (3), Advance messages conflict with every other messages. Hence, processes in g receive them in a total order. From our assumption in Section 5.1, write messages (i.e., messages carrying a write) are also received in the same order in g. Thus processes may only disagree on the order of read messages. That is, between two totally-ordered messages, they may disagree on the order of read messages. Since these messages do not conflict among each other, they change the clock only once at a process. Consequenly, updates on the clock follow the same sequence in a consensus group and the timestamp proposals are the same.

In more formal terms, let λ be some sequence of Begin and Advance messages. We write $[\lambda]$ the equivalence class of λ induced by the conflict relation ($\sim_{gb}$). That is, $\sigma \in [\lambda]$ if and only if $x \in \lambda \iff x \in \sigma$ and for any $x \sim_{gb} y$, $x <_\lambda y \iff x <_\sigma y$. The prefix relation over message sequences induces an order on their respective classes as follows: $[\sigma] \sqsubseteq [\lambda]$ if and only for some sequence of messages γ, $[\sigma\gamma] = [\lambda]$. For some process $p \in g$, let us write λ_p the sequence delivered at process p by Generic Broadcasts and $clk(M,p)$ the clock value at p after it processes a (Begin or Advance) message M. Generic Broadcast ensures that for any p and q in the same consensus group g, $[\lambda_p] \sqsubseteq [\lambda_q]$, or the converse, hold (see Lamport [2005]). We establish that if $[\lambda_p] \sqsubseteq [\lambda_q]$ then for any $M \in \lambda_p$, $clk(M,p) = clk(M,q)$. Invariant Base 4 follows from this property.

The proof of the property is by induction on the messages in λ_p. Consider that it holds until some $M \in \lambda_p$. Let $N <_{\lambda_p} M$ be the last message to change the clock K at p before M (i.e., due to blocks in lines 11-12 and lines 25-26). For some λ, λ' and λ'', $\lambda_p = \lambda N \lambda' M \lambda''$. Notice that if λ' is non-empty then λ' contains only read messages M' with $clk(M',p) = clk(N,p)$. (case $N <_{\lambda_q} M$) Let us write $\lambda_q = \sigma N \sigma' M \sigma''$, for some appropriate sequences σ, σ', and σ''. Suppose $M' \in \sigma'$. If M' is a write or an Advance message, then $N <_{\lambda_q} M' <_{\lambda_q} M$, but $M' \notin \lambda'$. It follows that $[\lambda_p] \not\sqsubseteq [\lambda_q]$. Thus, σ' only contains reads and for any such message M', $clk(M') = clk(N)$. By our induction hypothesis, $clk(N,p) = clk(N,q)$. (case $M <_{\lambda_q} N$) Similarly, we can write $\lambda_q = \sigma M \sigma' N \sigma'''$. If M (or N) conveys a write or is an Advance message then $[\lambda_p] \not\sqsubseteq [\lambda_q]$. Thus, M and N must be read messages. The same is true in case σ' does not contain only read messages. Then, observe that $clk(N,p) = clk(M,p)$ and $clk(M,q) = clk(N,q)$. We conclude using $clk(N,p) = clk(N,q)$, from our induction hypothesis. □

Proof Invariant SAFa. Consider a message m. According to the assumptions in Section 5.1, $m.d$ is partitioned uniquely into a set $\{g_1, \ldots, g_m\}$ of consensus groups. A process decides the final timestamp of a message at line 19. This happens when for each group g_i in the partition, p has a timestamp proposal from g_i (line 17). Hence, because of Invariant Base 4, processes must agree on the same final timestamp. □

Proof Invariant SAFb. Assume that a process p delivers a message m with a timestamp t. Let m' be some message conflicting with m, also addressed to p, whose final timestamp is t' (by Invariant SAFa processes agree on this), with $(m',t') < (m,t)$. Below, we establish that m' is already delivered at the time p delivers m.

Consider the point in time τ where p delivers m. The block in lines 27 to 28 is responsible for the delivery. In particular, this block has a guard to ensure that things happen in the right order (line 27). Namely, m is delivered with timestamp t only if for any message m' in Mem either m does not conflict with m', or m has a lower timestamp than m'.

If m' is already delivered at time τ, we are done. Otherwise, assume that m' is delivered later, or not delivered at all by p. Below, we prove that a contradiction is reached.

Message m' conflicts with m and $(m',t') < (m,t)$. Thus at time τ, either m' is in Mem at p with a higher timestamp $t'' > t$, or m' is not in Mem. In the former case, Mem contains a $\texttt{Propose}(m',t'')$. In light of the pseudo-code of Algorithm 2, necessarily $t' \geq t'' > t$; contradiction. Alternatively, consider the case where m' is absent from Mem at τ. Let t'' be the timestamp assigned at line 14 by process p to m'. Necessarily, $t'' \leq t' < t$. This event must happen before time τ due to the precondition in line 27, requiring $K \geq t$. It follows that there exists a $\texttt{Propose}(m,t'')$ in Mem at an earlier time than τ. Because m' is not delivered yet, this message is still in Mem or by Invariant Base 4, there is a $\texttt{Deliver}(m',t')$. In both cases, we reach the desired contradiction. □

Proof Invariant SAF. The proof follows from Invariant SAFa and Invariant SAFb. From Invariant SAFa, the processes in the destination agree on every message's final timestamp. Invariant SAFb guarantees delivery with a deterministic agreed order, utilizing the timestamp or the message's identifier to break ties.

With more details, the proof is by contradiction and as follows: Assume m and m' are delivered with respectively timestamp t and t' in a run. Consider that $(m',t') < (m,t)$ and that for some process p, $m \rightarrow_p m'$. Let t'' be the timestamp used by p to deliver m. Applying Invariant SAFa, $t'' = t$. By Invariant SAFb, because $(m',t') < (m,t)$ and $p \in m'.d$,

p must deliver (m', t') before m. Hence, $m' \rightarrow_p m$ and we have a contradiction by Invariant Base 1. □

Theorem 5.1. *Algorithm 2 guarantees the Ordering property of generic multicast.*

Proof. For the sake of contradiction, assume that Algorithm 2 violates Ordering. By definition, there exists a cycle in the delivery of messages across the system. By Invariant Base 1, this cycle contains at least two messages. In other words, for some $k \geq 1$, there exist messages $m_0, \ldots, m_k$ and processes $q_0, \ldots, q_k$ such that $m_0 \rightarrow_{q_0} m_1 \rightarrow_{q_1} \ldots \rightarrow_{q_{k-1}} m_k \rightarrow_{q_k} m_0$. For any $i \in [0, k]$, let t_i be the timestamp of m_i when the message is delivered at process p_i. Applying Invariant SAF, $(m_i, t_i) < (m_{i+1\ [k]}, t_{i+1\ [k]})$. Hence, $(m_0, t_0) < (m_0, t_0)$; contradiction. □

5.4.2 Liveness

We consider the following invariants:

LIVa If a correct process GM-Send(m) or a process GB-Deliver(Begin(m)), eventually all correct processes in $m.d$ insert a Deliver(m, _) in Mem.

LIVb If a correct process p adds Deliver(m, t) in the Mem set, eventually, p's clock is equal to or higher than t.

LIVc If a correct process p includes a Deliver(m, _) in Mem, then p eventually GM-Deliver(m).

Proof Invariant LIVa. The critical lines are the for-loop in lines 7 and 8. This procedure might have two outcomes. Either every group in m's destination delivers a Begin(m) message, or not.

For starters, consider the former case. At this stage, incorrect process crashes does not affect liveness as we assume each group has at least one correct process. In detail, since each group contains one correct process, they share proposals at lines 15 and 16. These processes being correct and the channel reliable, the delivery of Propose(m, _) messages from each group in $m.d$ eventually takes place at the correct processes in $m.d$. Thus, each such process calculates the final timestamp at line 18. Then, it adds a Deliver(m, _) message to Mem at line 19, which concludes the proof.

Alternatively, consider the second case, that is some group does not deliver a Begin(m) message. We observe that this case cannot happen if the sender is correct. Hence, from the assumptions in Invariant LIVa, a process q executes GB-Deliver(Begin(m)). Let $g \subseteq m.d$ be its group. Group g contains at least one correct process; name it p. From the Termination property of generic broadcast, p delivers Begin(m) because q did. After delivering Begin(m) at line 9, p submits its proposal to every other processes in $m.d$ then adds Propose(m, _) to Mem at line 14. Since the sender process crashes, process p's failure detector ($\mathcal{D}$) eventually suspects $m.src$, triggering the recovery procedure (Algorithm 3). Therefore, as p is correct, it successfully broadcasts a Begin(m) message to every other group in $m.d$ (which has not proposed a timestamp to m). Then, we may close the proof using the first case above. □

Proof Invariant LIVb. Invariant LIVa guarantees that each correct processes in $m.d$ eventually inserts a Deliver(m, t) message in Mem. After deciding that t is the final timestamp of m, a process verifies if the clock needs synchronization at line 20. There are two outcomes for this, namely the timestamp is higher or not. In the former case, the proof is over. In the latter, p's local group must synchronize its clock. To this end, p broadcasts an Advance(t) message to the group. As p is correct, Advance(t) is eventually delivered at line 23. Then, the clock is bumped (if needed) to a higher value than t at line 25. □

Proof Invariant LIVc. First of all, let us note $\mathcal{T}$ the global (discrete) time of the distributed system. We consider the following potential function Φ in $\mathcal{T} \times \Pi \mapsto 2^{\mathcal{M} \times \mathbb{N}}$: given a time $\tau \in \mathcal{T}$ and some process $p \in \Pi$, $\Phi(\tau, p)$ returns all the pairs (m, t) such that message m is stored in Mem at p with timestamp t (i.e., Mem contains a Deliver(m, t) or Propose(m, t) message.) Then, for such a set of pairs, $\phi(\tau, p, ts_f)$ are all the messages with a timestamp smaller than ts_f.

Assume a process p inserts Deliver(m, ts_f) in Mem. Applying Invariant LIVb, the clock of p eventually passes ts_f. Let τ be the point in time when this happens. By Invariant SAFb, $\phi(\tau', p, ts_f)$ is a decreasing function for any later point $\tau' > \tau$ in time.

Now, assume that $\phi(\tau, p, ts_f)$ is empty. The precondition at line 27 for m is true at time τ. Indeed, all the messages m' preceding m would be in $\phi(\tau, p, ts_f)$ which is by assumption empty. Moreover, it must be always true at any later point in time. Hence, p delivers eventually message m.

Otherwise, if $\phi(t, p, ts_f)$ is not empty, we can apply inductively the above reasoning on every message in $\phi(t, p, ts_f)$ starting from its smallest element. Thus, from what precedes, function $\phi(t, p, ts_f)$ converges towards an empty set over time and message m is eventually delivered. □

Theorem 5.2. *Algorithm 2 guarantees the Termination property of generic multicast.*

Proof. Follows from Invariant LIVa, Invariant LIVb, and Invariant LIVc. □

5.5 Handling Arbitrary Conflicts

Algorithm 2 is limited to read/write conflicts over a single data item. In this section, we adjust our solution to handle the other cases. First, we illustrate the problem that may arise in Algorithm 2 with an arbitrary conflict relation. Then, we present changes to accommodate a conflict relation partitioned into several disjoint classes.

Limitations Some assumptions on the conflict relation ($\sim$) are needed for Algorithm 2 to be safe. To illustrate, consider the following scenario: Let g be a (consensus) group with processes p and q. Three messages are submitted to Algorithm 2, m_1, m_2, and m_3, all addressed to destination groups that include g. Message m_1 commutes with the other two messages, while m_2 and m_3 conflict. Assume that process p delivers through Generic Broadcast, in this order, Begin(m_1), Begin(m_2), and Begin(m_3). Hence, it proposes timestamp

Algorithm 4 Multiple conflict classes

```
 1: Variables:
 2: $K[\overline{m}] \leftarrow \lambda\overline{m}.0$
 3: $Mem \leftarrow \emptyset$
 4: $PreviousMsgs[\overline{m}] \leftarrow \lambda\overline{m}.\emptyset$
    ...
 9: when GB-Deliver$_g$(Begin (m))
10:     if $\exists\, m' \in PreviousMsgs[\overline{m}] : m \sim m'$ then
11:         $K[\overline{m} \leftarrow K[\overline{m}] + 1$
12:         $PreviousMsgs[\overline{m}] \leftarrow \emptyset$
13:     $PreviousMsgs[\overline{m}] \leftarrow PreviousMsgs[\overline{m}] \cup \{m\}$
14:     $Mem \leftarrow Mem \cup \texttt{Propose}(m, K[\overline{m}])$
15:     for all $q \in m.d$ do
16:         $Send\langle \texttt{Propose}(m, K[\overline{m}]), q\rangle$
17: when $Receive\langle \text{PROPOSE}(m, \_), \_\rangle$
    ...
20:     if $K[\overline{m}] < ts_f$ then
21:         let $g \in \Gamma$ such that $p \in g$
22:         GB-Send$_g$($\texttt{Advance}(ts_f, \overline{m})$)
23: when GB-Deliver$_g$($\text{ADVANCE}(t, \overline{m})$)
24:     if $t > K[\overline{m}]$ then
25:         $K[\overline{m}] \leftarrow t$
26:         $PreviousMsgs[\overline{m}] \leftarrow \emptyset$
    ...
```

0 for m_1, 0 for m_2, and 1 for m_3. Meanwhile, process q delivers Begin(m_2), Begin(m_3), and Begin(m_1). According to Equation (3), such a delivery is permitted by Generic Broadcast. This leads q to propose 0 for m_2, 1 for m_3 and 1 for m_1. In this scenario, p and q report different timestamp proposals for message m_1 while being in the same consensus group g. This breaks Invariant Base 4 and may lead processes to deliver m_1 in incompatible orders.

Multiple Conflict Classes The aformentionned issue happens because m_1 is not in the same conflict class as m_2 and m_3. Splitting the timestamping mechanism of Algorithm 2 per class solves the problem. Algorithm 4 implements such an approach. It follows the logic of Algorithm 2 and handles any conflict relation that partitions into a set of read/write conflict classes.

In detail, a conflict class is *read/write* when each operation is either a read or a write, and only reads do not conflict. We assume that $\sim$ (or some safe approximation of it) partitions into such classes. For instance, in the case of a key-value store (Section 3.3), keys are segmented. Operations can be batched, but in a batch they must all access the same segment. Several storage systems, e.g., Apache Cassandra [Lakshman and Malik, 2010], have this restriction.

For some message m, $\overline{m}$ defines its conflict class. Algorithm 4 uses one clock and one set to track conflicts per class (lines 2 to 4). Initially, for some class $\overline{m}$, $K[\overline{m}]$ is set to 0 and $PreviousMsgs[\overline{m}]$ is empty. Variable Mem spans all the classes, and it plays the same role as previously.

Algorithm 4 details the changes made to Algorithm 2. For instance, in lines 10 to 12, the algorithm updates the class variables in case a conflict is detected. In short, the algorithm follows the same logic as Algorithm 2, applying it per conflict class. Its correctness can be established using the reasoning and invariants in Section 5.4.

5.6 Performance

Algorithm 2 ensures the two properties (Rigidness and Minimality) listed in Section 3.4. In what follows, we prove that the algorithm is matching the performance listed in Table 1. Performance is given for non-faulty runs, that is, runs during which there are no failures and no failure suspicions. In practice, this corresponds to the common case for real systems. Finally, we conclude the section with a discussion about metadata management.

When measuring latency, we consider that Algorithm 2 makes use of the fastest generic broadcast known to date (*e.g.*, [Ryabinin *et al.*, 2024; Park and Ousterhout, 2019; Sutra and Shapiro, 2011]). Such algorithms ensure that in a failure-free run, a message is delivered after two message delays if there are no concurrent conflicting messages and three otherwise.

Hereafter, we write R the set of failure-free runs for Algorithm 2. Among these runs, $R_{cf} \subset R$ (respectively, $R_{co} \subset R$) are the conflict-free (resp., contention-free) ones. Notice that there is no ordering relation between R_{cf} and R_{co}. We examine in order each of these classes to establish the results in Table 1.

Conflict-free For starters, we consider the class of conflict-free runs. This class is illustrated in Figure 2 where g_1 delivers the message after just 3 message delays: The blue box for generic broadcast takes 2 message delays. It is followed with the exchange of timestamp proposals, then delivery of the message at g_1.

For some run r, we note $dl_r(m)$ the delivery latency of a message m in r. We can establish the following result.

Proposition 5.1. *In every run $r \in R_{cf}$, $dl_r(m) \leq 3$ for any m.*

Proof. After two message delays, each process in $m.d$ delivers Begin(m) at line 9. There is no conflict in run r. Hence, variable K equals 0, and the condition at line 10 is false. After an additional message delay, the timestamp proposals are exchanged at lines 15 to 17. Upon collecting such proposals, the final timestamp $ts_f = 0$ is known at line 19. Hence, Deliver($m, 0$) is added to Mem. No message conflicts with m. Thus, the precondition at line 27 is valid, and m is delivered after 3 message delays. □

Collision-free Recall from Section 3.5 that a collision-free run is a run in which no two messages are sent concurrently to the same location. This means that when a message m is multicast, any message m' having a common process $p \in m.d \cap m'.d$ is delivered at that process p. In [Gotsman *et al.*, 2019], the authors introduce the notion of *commit latency* for Skeen-like algorithms. The commit latency measures the time it takes for a message to get assigned a final timestamp. For collision-free runs, the commit latency corresponds to the delivery latency (Theorem 3 in [Gotsman *et al.*, 2019]). Proposition 5.2 establishes that Algorithm 2 takes five message delays to commit a message in a collision-free run.

Proposition 5.2. *In every run $r \in R_{co}$, $dl_r(m) \leq 5$ for any m.*

Proof. The proof starts similarly to the proof of Proposition 5.1. At a process, the final timestamp of a message m is known after three message delays: Generic broadcast ensures that $\texttt{Begin}(m)$ is delivered after two message delays (because the run is collision-free). Then, we add one more message delay due to the exchange of timestamp proposals. Let t be the final timestamp of m. To deliver m, the precondition at line 27 must be true. This precondition demands that the logical clock (K) is higher than t. Once this holds, message m is delivered right away because there are no pending messages locally (as the run is collision-free). According to the pseudo-code at line 20, either the precondition is true at the time $\texttt{Deliver}(m, t)$ is added to Mem (line 19), or an $\texttt{Advance}(t)$ message is generic broadcast (line 22). In the latter case, after two more message delays, the code in lines 23 to 26 triggers. This ensures that the precondition at line 27 is true later on. □

To illustrate the above result, let us go back to Figure 2. This figure depicts a collision-free scenario for Algorithm 2. In particular, because g_2 has a clock smaller than the decided timestamp (42), it needs to bump its clock. This computation happens using generic broadcast (second green box in Figure 2) and takes two more message delays. Once the clock is bumped (bottom of the figure), the message is delivered. In total, it takes g_2 five message delays to deliver the message in this scenario.

Failure-free The failure-free latency is defined as the clock update latency plus the commit latency in failure-free runs [Gotsman *et al.*, 2019]. Lemma 5.3 proves that the commit latency of a message is seven for this class of runs.

Lemma 5.3. *For any $r \in R$, the commit latency in r is 7.*

Proof. The proof is almost identical to the one used in Proposition 5.2. The sole difference is the time it takes for generic broadcast to deliver a message. Here, we count three message delays instead of two. This comes from the possible contention in the primitive, leading to an additional message delay in the worst case. □

The *clock update latency* measures the (worst-case) number of message delays to wait before all the messages with a lower timestamp are known at the destination group once the final timestamp is computed. Equivalently, using the formulation in [Pacheco *et al.*, 2023a], this is the maximum delay after which no (consensus) group will assign another message a local timestamp smaller than the final timestamp. For Algorithm 2, Lemma 5.4 tells us that this demands waiting for four message delays.

Lemma 5.4. *For any $r \in R$, the clock-update latency in r is 4.*

Proof. Consider that some message m has a final timestamp t. If a message m' ends up with a lower timestamp than t, this message is delivered right before the clock is updated to (at least) t. In the worst case, m' is generic broadcast at this point by another (consensus) group, say g. It takes three message delays to get delivered at g. Then, the timestamp proposals are computed for m' and exchanged among the group partitioning the destination. Hence, four message delays were needed in total to compute the timestamp of m' since the clock was updated. □

In the light of the last two lemmas, we may conclude the following about the performance of Algorithm 2 in failure-free runs.

Corollary 5.4.1. *In every run $r \in R$, $dl_r(m) \leq 11$ for any m.*

6 Real-Time

This section motivates the interest of tracking real-time in generic multicast and explains how to implement it.

6.1 Motivation

As indicated in Section 3.3, data stores commonly provide an operator to access multiple items at once. Typically, such an operator is of the form `start...end`, permitting to group several operations in a batch. In this context, prior works [Pacheco *et al.*, 2023b; Bezerra *et al.*, 2014] observe that atomic multicast does not suffice *out-of-the-box* to maintain data consistency. This is also the case with generic multicast, as illustrated in Figure 3.

Figure 3a shows a run that breaks data consistency. The figure has two clients, c_1 and c_2. Client c_2 submits $w_1 = \texttt{write}(x, 10)$, waits for its completion, then submits $w_2 = \texttt{write}(y, 10)$. Concurrently, client c_1 executes a batch β of read operations to fetch keys x and y. The run in Figure 3a is not causally consistent: c_1 reads $x = 0$ and $y = 10$ despite that c_2 writes to x before accessing y.

Figure 3b explains how this run might happen with Algorithm 1. The key-value store has two replica processes, p and q. Process p replicates item x while process q replicates item y. Initially, $x = y = 0$ and p and q have their clocks K to the values 100 and 1, respectively.

Process p receives β and sends a `Propose` message to q. This message carries the timestamp 101 for β. Then, p receives w_1. Since this operation has a higher timestamp than β, its execution gets delayed.

Concurrently, q also receives β and sends a `Propose` message to p carrying timestamp 2. When this message is received by p, it delivers β with timestamp 101. The batch is executed locally and p returns $x = 0$ to client c_1. After this, process p executes w_1 and inform c_2.

Once its first operation is executed, client c_2 requests the second write. The operation is received by q. After this, the `Propose` message from p is received. Because the final timestamp of w_2 is 3, which is lower than β's, q executes w_2 before β. Consequently, q returns $y = 10$ to client c_1, breaking consistency.

6.2 Strict Ordering

In Figure 3b, the delivery order is $\beta \rightarrow_p w_1$ and $w_2 \rightarrow_q \beta$. This order is consistent with the conflict relation, but as observed, this does not suffice to guarantee data consistency.

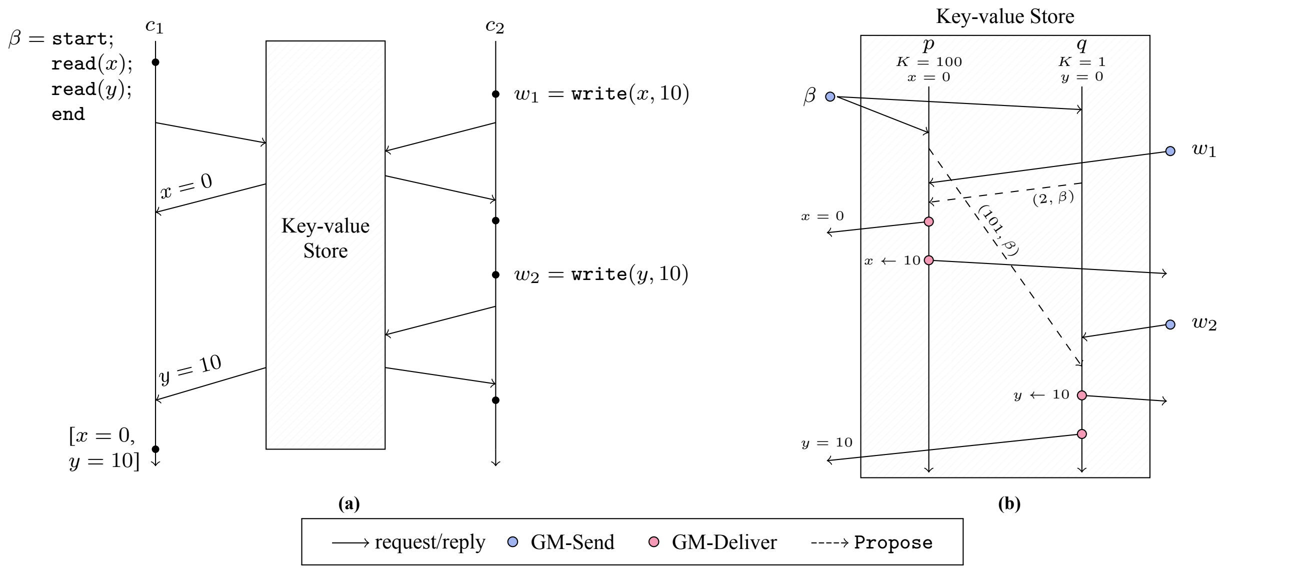

Figure 3. Generic multicast does not guarantee data consistency out-of-the-box when batched operations are permitted: **(a)** client c_1 sees the last version of y but not of x, yet c_2 updated x first; **(b)** illustrates how such a situation may occur with Algorithm 1.

The root cause of the problem is that generic multicast does not track the real-time order in which processes c_2 submits w_1, then w_2. Below, we strengthen the group communication primitive to enforce it.

Definition We write $m \rightsquigarrow m'$ when m is delivered at some destination in real-time before m' is multicast. This relation is called the real-time order. Generic multicast is *strict* when both the real-time and global orders are acyclic. Formally, let us note $\mapsto = \rightarrow \cup \rightsquigarrow$. This means that $m \mapsto m'$ holds when either a process delivers m before m' ($\rightarrow$), or m' is multicast after m is delivered at some destination ($\rightsquigarrow$). The Ordering property becomes:

- **Strict Ordering**: Relation $\mapsto$ is acyclic.

Notice that strictness is free when there is a single destination group. Indeed, if p delivers m before q broadcasts m', then necessarily $m \rightarrow_p m'$. This explains why atomic broadcast enforces linearizability out of the box.

Usage The run in Figure 3 is disallowed with strict ordering. In general, strict generic multicast would permit to implement partial state-machine replication (PSMR). PSMR is a variation of state-machine replication where processes may not replicate all the data available in the system [Enes *et al.*, 2021].

6.3 Implementation

In what follows, we adjust Algorithm 2 to implement strict genuine multicast.

Modifications. To enforce a strict ordering in Algorithm 2, we add an extra exchange across consensus groups when a message is ready for delivery. This is similar to prior works in literature, e.g., [Bezerra *et al.*, 2014; Enes *et al.*, 2021; Pacheco *et al.*, 2023b]. The exchange only happens when several consensus groups are involved.

Algorithm 5 details the modifications to Algorithm 2. We use a Ready message to signal when a consensus group is ready for delivery (lines 28 to 30). Processes await one Ready from each consensus group (line 31) before delivering. This ensures that once the message is delivered, any conflicting messages later multicast must have a higher timestamp. The rest of the algorithm does not change.

Algorithm 5 Strict generic multicast.

```
...
28: procedure ready(m)
      pre: $\wedge\ \exists t \leq K$ : Deliver$(m,t) \in Mem$
             $\wedge\ \forall m', t'$ : Propose|Deliver$(m',t') \in Mem$
                  $\implies (m \not\sim m' \vee (m,t) < (m',t'))$
29:     for all $q \in m.d$ do
30:         $Send\langle$Ready$(m), q\rangle$
31: procedure GM-Deliver(m)
      pre: $m.d = \cup\{g \in \Gamma \mid \exists q \in g : Receive\langle$Ready$(m), q\rangle\}$
32:     $Mem \leftarrow Mem \setminus$ Deliver$(m,t)$
```

Correctness. The invariants listed in Section 5.4 still hold with Algorithm 5. In particular, processes agree on the final timestamps of messages and they deliver them accordingly to this order. We add and prove the following new invariant:

SAFs If $m \mapsto m'$ then whenever some process p delivers m', some process has already delivered m.

Proof Invariant SAFs. From the definition of $\mapsto$, there are two cases to consider: either *(i)* $m \rightsquigarrow m'$, or *(ii)* $m \rightarrow m'$. We examine each of them below.

Case (i). The invariant trivially holds by the definition of $m \rightsquigarrow m'$. Indeed, $m \rightsquigarrow m'$ captures that when m' is disseminated, m is delivered at a process. Process p necessarily delivers m' after such an event occurs.

Case (ii). Relation $m \rightarrow m'$ implies that m and m' have common consensus groups. We write $\mathcal{G} = m.d \cap m'.d$ such groups and let g be one of them. Let τ be the time at which p delivers m'. For this to happen, p must receive a Ready(m') message from consensus group g. Hence at time $\tau' < \tau$ some process $q \in g$ sends a Ready(m') message to p. Because $m \rightarrow m'$, invariant SAF guarantees that $(m,t) < (m',t')$, where t and t' are respectively the final timestamps of m and m'. Consequently, at time τ, either q has already delivered

m or it is stored in variable Mem. (Message m cannot be unknown at q because by Invariant Base 2, t would be higher than t'.) In the former case, we are done. In the later, the second precondition at line 28 prevents q to send a message to p; contradiction. □

Theorem 6.1. *Algorithm 5 implements strict generic multicast.*

Proof. For the sake of contradiction, assume that Algorithm 5 violates Strict Ordering. By definition, there is a cycle in the delivered messages across the system. This means that for some $k \geq 1$, there exist messages $m_1, \ldots, m_k$, such that $m_1 \mapsto \ldots \mapsto m_k \mapsto m_1$. Consider the time τ at which a process first delivers m_1. Applying Invariant SAFs inductively to the cycle above, some process delivers message m_1 before time τ; a contradiction.

All the other properties of the group communication primitive follow from the same reasoning as in Section 5.4. □

7 Closing Remarks

This section discusses our results and some possible extensions before closing.

About genericity Regarding generic group communication primitives, we note that they might not always be the solution to every problem performance-wise. For instance, simpler algorithmic solutions can be more efficient in specific scenarios (e.g., RDMA networks [Wang *et al.*, 2017]), or when it could be faster to broadcast messages instead of multicasting them [Schiper *et al.*, 2009]. This comes from the fact that group communication primitives are sensitive to the considered application usage.

Future work We conjecture that it is possible to cut one message delay in Algorithm 2. The solution would be similar in spirit to PrimCast [Pacheco *et al.*, 2023a]: each consensus group listens to the decisions taken by the other groups at the destination. Upon receiving a quorum of commit acknowledgments (2B messages in Generalized Paxos [Lamport, 2005]), a process knows the clock of a remote group right away. This can be implemented with the notion of witness as found in state-machine replication algorithms (*e.g.*, [Hunt *et al.*, 2010; Park and Ousterhout, 2019]). This optimization skips the need to exchange Propose messages in Algorithm 2.

Conclusion This work defines generic multicast in crash-prone distributed systems. It presents two matching solutions that are variations of the timestamping approach invented by Skeen. The first solution works in a failure-free environment. It is extended into a failure-prone algorithm using the standard partitioning assumption over destination groups. By employing techniques from other well-established works in the literature, the resulting algorithm (and recovery procedure) is relatively simple and understandable. The algorithm uses a generic broadcast in each group to compute timestamp proposals and deliver messages in a consistent order. When the run is conflict-free, that is, no two messages conflict, the algorithm delivers each message after three message delays. We also detail a variation that delivers messages across the system in an order consistent with real-time, at the cost of a message delay.

As distributed applications grow in complexity and scale, group communication primitives are increasingly valuable to implement coordination and fault-tolerance. Generic multicast is a new, flexible group communication primitive that covers the full spectrum of previous reliable and atomic primitives. Depending on specific application needs, it can be adjusted to tailor an implementation to a given context while offering strong ordering guarantees.

Declarations

Funding

This work was supported by the CHIST-ERA European initiative (grant CHIST-ERA-22-SPiDDS-05, REDONDA project), and by Agence Nationale de la Recherche (grant ANR-23-CHR4-0009).

Authors' Contributions

This paper is an extended version of previous research [Bolina *et al.*, 2024], started by Douglas Antunes and later improved and expanded by José Bolina, under the supervision of Lásaro Camargos. The work was refined and published with the important participation of Pierre Sutra. This extended work presented here is due to José Bolina and Pierre Sutra.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

Data can be made available upon request.

References

Ahmed-Nacer, T., Sutra, P., and Conan, D. (2016). The convoy effect in atomic multicast. In *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*, pages 67–72. IEEE. DOI: 10.1109/SRDSW.2016.22.

Amazon (2008). Aws simple storage service. Available at: `https://aws.amazon.com/s3`. Accessed: 2025-08-18.

Benz, S., Marandi, P. J., Pedone, F., and Garbinato, B. (2014). Building global and scalable systems with atomic multicast. In *Proceedings of the 15th International Middleware Conference*, Middleware, page 169–180, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/2663165.2663323.

Bezerra, C. E. B., Pedone, F., and van Renesse, R. (2014). Scalable state-machine replication. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, pages 331–342. IEEE Computer Society. DOI: 10.1109/DSN.2014.41.

Birman, K. P. and Joseph, T. A. (1987). Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76. DOI: 10.1145/7351.7478.

Bolina, J., Sutra, P., Antunes, D., and Camargos, L. (2024). Generic multicast. In *Proceedings of the 13th Latin-American Symposium on Dependable and Secure Computing*, LADC '24, page 81–90, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3697090.3697095.

Camaioni, M., Guerraoui, R., Monti, M., Roman, P., Vidigueira, M., and Voron, G. (2024). Chop chop: Byzantine atomic broadcast to the network limit. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 269–287. USENIX Association. DOI: 10.48550/arxiv.2304.07081.

Chandra, T. D., Griesemer, R., and Redstone, J. (2007). Paxos made live: an engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, page 398–407, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/1281100.1281103.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267. DOI: 10.1145/226643.226647.

Coelho, P. R., Schiper, N., and Pedone, F. (2017). Fast atomic multicast. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 37–48. DOI: 10.1109/DSN.2017.15.

Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. (2013). Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3). DOI: 10.1145/2491245.

Cowling, J. and Liskov, B. (2012). Granola: low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 21, USA. USENIX Association. Available at: `https://www.usenix.org/conference/atc12/technical-sessions/presentation/cowling`.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421. DOI: 10.1145/1041680.1041682.

Delporte-Gallet, C. and Fauconnier, H. (2000). Fault-tolerant genuine atomic multicast to multiple groups. In Butelle, F., editor, *Procedings of the 4th International Conference on Principles of Distributed Systems, OPODIS 2000, Paris, France, December 20-22, 2000*, Studia Informatica Universalis, pages 107–122. Suger, rue Catulienne Saint-Denis France. Available at:
.

Enes, V., Baquero, C., Gotsman, A., and Sutra, P. (2021). Efficient replication via timestamp stability. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 178–193, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3447786.3456236.

Fritzke, U., Ingels, P., Mostefaoui, A., and Raynal, M. (1998). Fault-tolerant total order multicast to asynchronous groups. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No.98CB36281)*, pages 228–234. DOI: 10.1109/RELDIS.1998.740503.

Gotsman, A., Lefort, A., and Chockler, G. (2019). White-box atomic multicast. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 176–187. DOI: 10.1109/DSN.2019.00030.

Gray, J. (1978). Notes on data base operating systems. In *Operating Systems, An Advanced Course*, page 393–481, Berlin, Heidelberg. Springer-Verlag. DOI: 10.1007/3-540-08755-9_9.

Guerraoui, R. and Schiper, A. (1997). Genuine atomic multicast. In *Distributed Algorithms, 11th International Workshop, WDAG '97, Saarbrücken, Germany, September 24-26, 1997, Proceedings*, volume 1320 of *Lecture Notes in Computer Science*, pages 141–154. Springer. DOI: 10.1007/BFB0030681.

Guerraoui, R. and Schiper, A. (2001). Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 254(1):297–316. DOI: 10.1016/S0304-3975(99)00161-9.

Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 11, USA. USENIX Association. Available at:`https://www.usenix.org/legacy/events/atc10/tech/full_papers/Hunt.pdf`.

Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. 44(2):35–40. DOI: 10.1145/1773912.1773922.

Lamport, L. (2005). Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research. Available at:`https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/`.

Lamport, L. (2006). Lower bounds for asynchronous consensus. *Distributed Comput.*, 19(2):104–125. DOI: 10.1007/S00446-006-0155-X.

Lampson, B. W. and Sturgis, H. E. (1979). *Crash recovery in a distributed data storage system*. Xerox Palo Alto Research Center Palo Alto, California. Available at:`https://web.cs.wpi.edu/~cs502/cisco11/Papers/LampsonSturgis_Crash%20recovery_later.pdf`.

Moraru, I., Andersen, D. G., and Kaminsky, M. (2013). There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 358–372, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/2517349.2517350.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX An-*

nual Technical Conference, USENIX ATC'14, page 305–320, USA. USENIX Association. Available at:`https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

Pacheco, L., Coelho, P., and Pedone, F. (2023a). Primcast: A latency-efficient atomic multicast. In *Proceedings of the 24th International Middleware Conference*, Middleware '23, page 124–136, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3590140.3629110.

Pacheco, L., Dotti, F., and Pedone, F. (2023b). Strengthening atomic multicast for partitioned state machine replication. In *Proceedings of the 11th Latin-American Symposium on Dependable Computing*, LADC '22, page 51–60, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3569902.3569909.

Park, S. J. and Ousterhout, J. (2019). Exploiting commutativity for practical fast replication. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 47–64, USA. USENIX Association. Available at:`https://www.usenix.org/conference/nsdi19/presentation/park`.

Pedone, F. and Schiper, A. (1999). Generic broadcast. In *International Symposium on Distributed Computing (PODC)*, pages 94–106. Springer. DOI: 10.1007/3-540-48169-9_7.

Pedone, F. and Schiper, A. (2002). Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107. DOI: 10.1007/s004460100061.

Ryabinin, F., Gotsman, A., and Sutra, P. (2024). Swiftpaxos: fast geo-replicated state machines. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI'24, USA. USENIX Association. Available at:`https://www.usenix.org/conference/nsdi24/presentation/ryabinin`.

Schiper, N. (2009). *On Multicast Primitives in Large Networks and Partial Replication Protocols*. PhD thesis. Available at:`https://sonar.ch/global/documents/318203`.

Schiper, N. and Pedone, F. (2007). Optimal atomic broadcast and multicast algorithms for wide area networks. DOI: 10.1145/1281100.1281185.

Schiper, N., Sutra, P., and Pedone, F. (2009). Genuine versus non-genuine atomic multicast protocols for wide area networks: An empirical study. In *2009 28th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 166–175. IEEE. DOI: 10.1109/SRDS.2009.12.

Sutra, P. (2022). The weakest failure detector for genuine atomic multicast. In *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*, volume 246 of *LIPIcs*, pages 35:1–35:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. DOI: 10.4230/LIPICS.DISC.2022.35.

Sutra, P. and Shapiro, M. (2011). Fast genuine generalized consensus. In *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, pages 255–264. DOI: 10.1109/SRDS.2011.38.

The etcd Authors (2014). etcd. Available at:`https://etcd.io`. Accessed: 2025-08-18.

Wang, C., Jiang, J., Chen, X., Yi, N., and Cui, H. (2017). Apus: fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC, page 94–107, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3127479.3128609.