



HAL
open science

Generic multicast

Jose Bolina, Pierre Sutra, Douglas Antunes, Lasaro Camargos

► **To cite this version:**

Jose Bolina, Pierre Sutra, Douglas Antunes, Lasaro Camargos. Generic multicast. LADC 2024 - 13th Latin-American Symposium on Dependable and Secure Computing, Nov 2024, Recife, Brazil. pp.81-90, 10.1145/3697090.3697095 . hal-04909501

HAL Id: hal-04909501

<https://hal.science/hal-04909501v1>

Submitted on 10 Feb 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Generic Multicast

Full research paper

Jose Bolina*
Red Hat, Inc.
Uberlândia, Brazil
jbolina@redhat.com

Pierre Sutra
Télécom SudParis
Paris, France
INRIA
Paris, France
pierre.sutra@telecom-sudparis.eu

Douglas Antunes*
Federal University of Uberlândia
Uberlândia, Brazil
douglasnr@ufu.br

Lasaro Camargos
Federal University of Uberlândia
Uberlândia, Brazil
Weilliptic Inc.
Uberlândia, Brazil
lasaro@weilliptic.com

Abstract

Communication primitives play a central role in modern computing. 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 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.

CCS Concepts

• **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Theory of computation** → **Distributed algorithms**.

*The main contributions of these authors were made while studying as MSc students at the Computer Science School of Federal University of Uberlândia, Brazil.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

LADC 2024, November 26–29, 2024, Recife, Brazil

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1740-6/24/11
<https://doi.org/10.1145/3697090.3697095>

Keywords

Consensus, Multicast, Broadcast, Generalized Consensus.

ACM Reference Format:

Jose Bolina, Pierre Sutra, Douglas Antunes, and Lasaro Camargos. 2024. Generic Multicast: Full research paper. In *13th Latin-American Symposium on Dependable and Secure Computing (LADC 2024)*, November 26–29, 2024, Recife, Brazil. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3697090.3697095>

1 Introduction

Atomic broadcast is a fundamental building block of modern computing infrastructures. This 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 [7, 8, 11, 21, 26]. 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 [2, 12].

The second common generalization of atomic broadcast is *generic broadcast* (aka., generalized consensus). This primitive is introduced in the work of Pedone and Schiper [30] and Lamport [22]. 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 [22, 25, 30].

Algorithm	Latency			generic?	metadata per msg.
	conflict -free	collision -free	failure -free		
FastCast [10]	8	4	8	×	$O(1)$
White-Box [17]	5+1	3+1	5+1	×	$O(1)$
PrimCast [27]	5	3	5	×	$O(1)$
Tempo [15]	4	5	13	✓	$O(m)$
Ahmed et al. [1] †	3	5		✓	$O(1)$
Algorithm 1 †	3	3	5	✓	$O(1)$
Algorithm 2	3	5	11	✓	$O(1)$

Table 1: Atomic multicast algorithms. (+1: additional delay to non-leader processes; †: not fault-tolerant).

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 two new solutions.

With more details, we claim the following contributions: (i) the definition of generic multicast for crash-prone distributed systems, (ii) a base solution that extends the timestamping approach of Skeen [4], (iii) building upon this, an understandable fully-fledged fault-tolerant generic multicast algorithm.

Our algorithms extend well-established mechanisms proposed in the literature (e.g., [1, 4, 16, 33]). Our base (non fault-tolerant) solution delivers a message in three communication steps when no two messages conflict. Tolerating failures attains the same lower bound, for a total of 3 communication steps. To the best of our knowledge, this is the first time this delivery latency has been attained in a conflict-free scenario (see Table 1 for a comparison with prior works). Additionally, by expanding well-established constructs, the algorithms remain simple and understandable. Furthermore, the two algorithms are thrifty in the amount of metadata they use. Regarding standard approaches in the field, they solely need a set to store non-conflicting messages and a scalar.

Outline. The remaining 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 fully-fledged fault-tolerant algorithm. Additionally, this section argues about the performance of the algorithm and its correctness. We close in Section 6 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. 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 [4]. In this work, the authors describe an unpublished algorithm by Dale Skeen that use 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 to the message this new priority. At a process, messages are delivered in the order of their priority, from lowest to highest.

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 [4] 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 detect accurately if a process in the destination group has failed. To deal with it, Guerraoui and Schiper [19] propose 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. [16] and Delporte-Gallet and Fauconnier [14]. 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 it uses 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. This delays delivery. Such a situation is due to contention, creating a convoy effect in the system [1]. To deal with it, Ahmed-Nacer et al. [1] propose 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 [22, 29]. These algorithms use the semantics of messages to deliver them earlier. Building upon this, Ahmed-Nacer et al. [1] define generic multicast as a variation of atomic multicast that understands the message semantics. However, the definition in [1] 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 in earlier works (e.g., [32]).

Recent Progress. To bump the clock, Guerraoui and Schiper [19] execute a second agreement per consensus group. Schiper and Pedone [33] 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. [10] 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

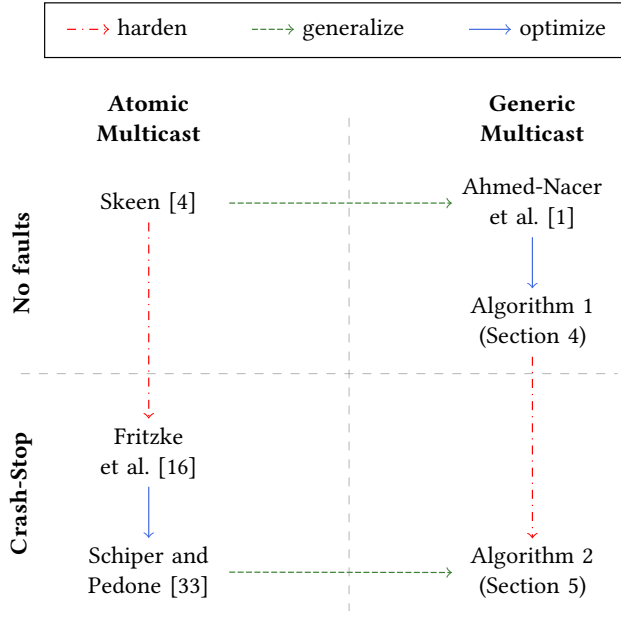


Figure 1: Structured derivations of generic multicast.

fast path correctly guesses the final timestamp. An improvement is made to the above schema by Gotsman et al. [17]. This algorithm opens the consensus “black box” to make additional optimizations. In the absence of collisions, the solution in [17] delivers a message in just three message delays at the leader of a consensus group. Non-leader members need an additional communication step. Prim-Cast [27] 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 [17]). 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 [15] leverages the semantics of messages. Messages are split into m classes (see Table 1). Two messages conflict, *i.e.*, do not commute, only if they are in the same class (identified with a key). When messages collide, but do not conflict, Tempo can deliver them in four message delays (“collision-free” column in Table 1).

The work in [35] 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 is possible. This 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 is taken into

account, as in generic broadcast, to order them only where necessary. Both effects contribute to better delivery performance, as illustrated in Table 1.

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.

3.1 System Model

The distributed system consists of a set $\Pi = \{p_1, p_2, \dots, p_{n \geq 2}\}$ of processes. A process may fail by crashing, halting its execution. When 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 $Send(m, p)$ sends the message m to the process p . The event $Receive(m, p)$ 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 [9]. 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 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 [22, 29]. Relation \sim is irreflexive and symmetric over \mathcal{M} . It is such that if two messages conflict, then they 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 $m \sim m'$, 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' . How messages are delivered across the system is captured with the *global delivery order*. 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 invokes GM-Send(m) or GM-Deliver(m) for 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 at all the processes in its destination group.

Flexibility. Generic multicast is a flexible communication primitive. If the conflict relation binds any two messages ($\sim = \mathcal{M} \times \mathcal{M}$), 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 [29]. 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 context 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). 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 begin...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 [11]. 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) [18, 24] to agree on committing or aborting its changes.

Atomic multicast provides an alternative design [2, 12]. 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 is a batch, an additional phase is needed, as detailed in [3].) 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 , isRead(c) evaluates to true when c is a read operation. We write 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. Formally,

$$c \sim d \triangleq \text{key}(c) = \text{key}(d) \wedge \neg (\text{isRead}(c) \wedge \text{isRead}(d)) \quad (1)$$

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

$$m \sim m' \triangleq \exists c, d \in \text{ops}(m) \times \text{ops}(m'). c \sim d \quad (2)$$

With generic multicast, operations on the key-value store that do not access the same items commute. Hence, this group communication primitive permits to order 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 re-visit 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 more messages than necessary. To avoid this, the authors in [29] introduce the notion of *strictness*. Such a property captures when an implementation permits non-conflicting messages to be delivered in different orders. The definition given in [29] is for generic broadcast. It is extended naturally to our context as follows:

(*Strictness*) 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 an execution ϵ during which where 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 [20], the authors introduce a minimality property that rules out such approaches.

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

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 [17], the authors define the notions of failure-free and collision-free latency. These metrics are defined when the system is stable, that is, when there are no failures and the system behaves synchronously. We now extend such metrics to the context of generic multicast. At core, we take into account the semantics of messages.

Runs are classified whether they contain concurrent *and conflicting* messages, or not.

The *delivery latency* of a message is the time interval between the moment it is multicast and delivered everywhere.¹ The *conflict-free latency* is the maximum 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.² Two messages m and m' are concurrent when neither m precedes m' nor the converse holds. The *collision-free latency* is the maximum delivery for a message when there is no concurrent message with a common destination. Last, the *failure-free latency* is the maximum 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 (see Section 3.3). The failure-free latency defines an upper bound on the time an operation takes to return. 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 (or a batch of them), 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 conflict-free latency.

Lower bounds. As mentioned earlier, atomic broadcast (and thus consensus) trivially reduces to generic multicast. Hence any known lower bound on this communication primitive applies to generic multicast. It is well-known that atomic 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 [22]. Now, when a collision occurs, three message delays are necessary [23].

4 A Base Solution

This section introduces a base solution to generic multicast. This algorithm follows the standard schema invented by Skeen [4]: For 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 needs only this to happen when a conflict occurs. Moreover, conflict-free messages are delivered in parallel without waiting in contrast to vanilla atomic multicast. Both mechanisms reduce the latency and 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 and the overall logic, then 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 that are ready for delivery.

¹In [17], the authors consider the moment the message is first delivered. Here, we follow the definition proposed in [27].

²The 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:      $Send(\text{Begin}(m), p)$ 
9:   WHEN  $Receive(\text{Begin}(m), q)$ 
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:       $Send(\text{Propose}(m, K), q)$ 
16:    WHEN  $\forall q \in m.d. Receive(\text{Propose}(m, ts), q)$ 
17:       $ts_f \leftarrow \max(\{ts : Receive(\text{Propose}(m, ts), q)\})$ 
18:      for all  $q \in m.d$  do
19:         $Send(\text{Deliver}(m, ts_f), q)$ 
20:    WHEN  $Receive(\text{Deliver}(m, ts_f), p)$ 
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( $m$ )
30:    pre:  $\wedge \exists t. (m, t) \in Delivering$ 
       $\wedge \forall m', t'. (m', t') \in Delivering \cup Pending$ 
       $\implies (m \neq m' \vee (m, t) < (m', t'))$ 
       $Delivering \leftarrow Delivering \setminus \{(m, t)\}$ 

```

- $PreviousMsgs$: 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.

Internals. To disseminate some message m , a process p invokes at line 6 operation $\text{GM-Send}(m)$. The operation sends a message $\text{Begin}(m)$ 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 $\text{Begin}(m)$ 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). The storage system would be linearizable taking full advantage of non-conflicting messages.

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., [16, 33]).

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.

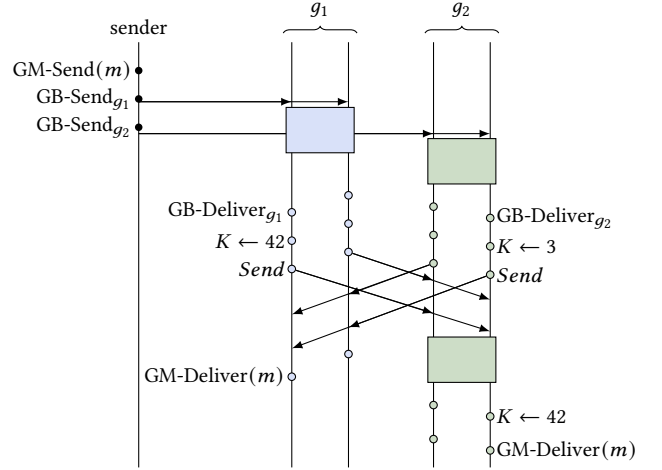


Figure 2: A run of Algorithm 2.

5.1 Additional Assumptions

Fault-tolerance. To be fault-tolerant, we need to strengthen our computational model. A standard assumption [13] is to assume that the system is partitionable into groups. Internally, each such group is robust enough to solve consensus. The destination a message is then defined as the union of one (or more) of these consensus groups.

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

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

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) abide by Equation (1). Later in the paper, we discuss this limitation and how it can be lifted.

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 employs 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 map stores the necessary information to process each message. It plays the joint roles of Pending and Delivering in Algorithm 1.

Algorithm 2 follows the standard transformation invented in [19]. 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 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 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 groups g_1 and g_2 partition the destination group of message $m.d$. 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 the blue box. In parallel g_2 also reaches an agreement on the proposal using the green box. Then, the two 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 group $g \in \mathcal{G}$, p broadcasts a $\text{Begin}(m)$ 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 the section.

Upon receiving a $\text{Begin}(m)$ 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 group g compute the exact same proposal for m . As a consequence, there is no need to disseminate the 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 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 a (consensus) group g is in 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 $\text{Advance}(ts_f)$ message within the local group.

Within a 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

Algorithm 2 Fault-tolerant generic multicast – code at p

```

1: Variables:
2:  $K \leftarrow 0$ 
3:  $\text{Mem} \leftarrow \emptyset$ 
4:  $\text{PreviousMsgs} \leftarrow \emptyset$ 

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 $_g(\text{Begin}(m))$ 

9:   WHEN GB-DELIVER $_g(\text{Begin}(m))$ 
10:    if  $\exists m' \in \text{PreviousMsgs}. m \sim m'$  then
11:       $K \leftarrow K + 1$ 
12:       $\text{PreviousMsgs} \leftarrow \emptyset$ 
13:       $\text{PreviousMsgs} \leftarrow \text{PreviousMsgs} \cup \{m\}$ 
14:       $\text{Mem} \leftarrow \text{Mem} \cup \text{Propose}(m, K)$ 
15:      for all  $q \in m.d \setminus g \cup \{p\}$  do
16:        Send(Propose( $m, K$ ),  $q$ )

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

24:   WHEN GB-DELIVER $_g(\text{Advance}(t))$ 
25:     if  $t > K$  then
26:        $K \leftarrow t$ 
27:        $\text{PreviousMsgs} \leftarrow \emptyset$ 

28:   PROCEDURE GM-DELIVER( $m$ )
29:     pre:  $\wedge \exists t \leq K. \text{Deliver}(m, t) \in \text{Mem}$ 
30:            $\wedge \forall m', t'. \text{Propose} | \text{Deliver}(m', t') \in \text{Mem}$ 
31:              $\implies (m \neq m' \vee (m, t) < (m', t'))$ 
32:      $\text{Mem} \leftarrow \text{Mem} \setminus \text{Deliver}(m, t)$ 

```

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 unity. Equation (3) specifies how this relation (\sim_{gb}) is set to satisfy the requirements for two (distinct) messages m and m' .

$$\begin{aligned}
m \sim_{gb} m' \triangleq & \vee m = \text{Advance}(_) \\
& \vee m' = \text{Advance}(_) \\
& \vee m = \text{Begin}(x) \wedge m' = \text{Begin}(y) \wedge x \sim y
\end{aligned} \tag{3}$$

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, that is message delivery occurs in the timestamp order.

Algorithm 3 Recovery mechanism

```

1: procedure RECOVER( $m$ )
   pre:  $\wedge \exists m. \text{Propose}(m, \_) \in \text{Mem}$ 
        $\wedge m.\text{src} \in \mathcal{D}$ 
2:   for all  $g \in \Gamma : g \subseteq m.d$  do
3:     if  $\nexists p \in g. \text{Receive}(\text{Propose}(m, \_), p)$  then
4:       GB-Send $_g(\text{Begin}(m))$ 

```

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 processes that are suspected to have failed [9]. 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 nor liveness.

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

5.4 Correctness

In what follows, we sketch the correctness of Algorithm 2. A complete proof appears in the long version of this paper [6]. Algorithm 2 was also verified using TLA⁺. The specification can be found online [5].

Safety. Integrity follows from the pseudo-code of Algorithm 2. In detail, a message $\text{Begin}(m)$ is delivered from generic broadcast at most once (line 9). Message m starts as a $\text{Propose}(m, _)$ message in variable Mem , then it is transformed into a $\text{Deliver}(m, _)$ message. Once m escapes variable Mem , it is delivered at the local process and never re-appears again.

Ordering is proved using the following safety invariant (SAF): For any two conflicting messages m and m' delivered in the run with respectively timestamps t and t' , if $m \rightarrow_p m'$ then $(m, t) < (m', t')$. To prove invariant (SAF), two intermediary invariants are obtained. First, (SAFa) processes in the destination group agree on the final timestamp of a message. This result is obtained by using the fact that (consensus) groups agree on the timestamp proposals. Second, (SAFb) a process delivers a message only if it has already delivered all the conflicting messages with a lower

timestamp. Such an invariant comes from the fact that the logical clock is monotonically growing, and that it ticks every time a new conflict is detected.

Liveness. Consider some message m that is either multicast by a correct process, or delivered somewhere. To prove termination (LIV), we decompose it into three secondary invariants. First, (LIVa) every correct process in $m.d$ eventually stores a $\text{Deliver}(m, _)$ message in variable Mem . Once this holds, invariant (LIVb) establishes that the clock of a correct process is eventually synchronized to m 's final timestamp. Invariant (LIVc) states that if a correct process stores a $\text{Deliver}(m, _)$ message in the Mem set, then it eventually delivers m . This third invariant is proved with the help of an appropriate potential function. The conjunction of LIVa, LIVb, and LIVc ensures that LIV holds.

5.5 Performance

Algorithm 2 ensures the two implementation properties (strictness 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., [28, 31, 36]). 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 $\text{Begin}(m)$ 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, $\text{Deliver}(m, 0)$ 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. \square

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 [17], 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 [17]). 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 one of Proposition 5.1. At a process, the final timestamp of a message m is known after three message delays: Generic broadcast ensures that $\text{Begin}(m)$ 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 $\text{Deliver}(m, t)$ is added to Mem (line 19), or an $\text{Advance}(t)$ 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. \square

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 [17]. 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. \square

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 [27], 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 to wait 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. \square

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

COROLLARY 5.5. *In every run $r \in R$, $dl_r(m) \leq 11$ for any m .*

6 Closing Remarks

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

About genericity. Algorithm 2 is limited to read/write conflicts over a single data item key. Nevertheless, it is possible to leverage commutativity in other situations. For instance, the very same algorithm works with two additional classes of operations: one in which operations conflict with nothing, and another where they conflict with everything. More complex situations demand replicas to store additional metadata. To track operations over different keys (or segments of keys), we can de-duplicate variables K and PreviousMsgs , using a pair of variables per key. Notice that Algorithm 1 does not have such a limitation, and it works for any conflict relation. We believe that this is due to an inherent trade-off between genericity and fault-tolerance.³

Regarding generic group communication primitives, we note that they might not always be the best approach. For instance, simpler algorithmic solutions can be more efficient in specific scenarios (e.g., RDMA networks [37]), or when it could be faster to broadcast messages instead of multicasting them [34]. This comes from the fact that group communication primitives are sensitive to the considered application usage.

Future work. The definition of generic multicast can be extended to capture real-time. In this case, an ordering relation binds two messages m and m' in case m is delivered at a process before m' is multicast (even if m and m' do not have a common process). Such a variation makes sense when the communication primitive is used in the context of data replication [3]. Tempo [15] ensures such a guarantee. Without it, a message can get delivered earlier in the algorithm after 3 message delays. We could extend Algorithm 2 to satisfy this additional property. In Algorithm 2, a process must know that the clock of its consensus group is after its final timestamp to deliver it. It suffices to change the precondition into the clocks of all the consensus groups in the destination.

In Table 1, the space usage of Tempo is higher than with our algorithms. This comes from the fact that this algorithm discriminates several classes of conflicts (typically using a key per conflict class). As outlined earlier, one may apply such an idea to Algorithm 2 at the cost of storing more metadata at each replica.

We conjecture that it is possible to cut one message delay in Algorithm 2. The solution would be similar in spirit to PrimCast [27]: Each consensus group listens to the decisions taken by the other

³Actually, it is possible to replace variable PreviousMsgs in Algorithm 2 with a singleton that simply stores the last operation (a read or a write).

groups at the destination. Upon receiving a quorum of commit acknowledgments (2B messages in Generalized Paxos [22]), 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., [21, 28]). 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) are 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.

References

- [1] 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.
- [2] Samuel Benz, Parisa Jalili Marandi, Fernando Pedone, and Benoît Garbinato. 2014. Building global and scalable systems with atomic multicast. In *Proceedings of the 15th International Middleware Conference (Bordeaux, France) (Middleware)*. Association for Computing Machinery, New York, NY, USA, 169–180. <https://doi.org/10.1145/2663165.2663323>
- [3] Carlos Eduardo Benevides Bezerra, Fernando Pedone, and Robbert van Renesse. 2014. Scalable State-Machine Replication. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*. IEEE Computer Society, 331–342. <https://doi.org/10.1109/DSN.2014.41>
- [4] 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.
- [5] José Augusto Bolina. 2022. <https://github.com/jabolina/mcast-tlaplus>
- [6] José Augusto Bolina, Pierre Sutra, Douglas Antunes Rocha, and Lasaro Camargos. 2024. Generic Multicast (Extended Version). <https://arxiv.org/abs/2410.01901>
- [7] Martina Camaioni, Rachid Guerraoui, Matteo Monti, Pierre-Louis Roman, Manuel Vidigueira, and Gauthier Voron. 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*, Ada Gavrilovska and Douglas B. Terry (Eds.). USENIX Association, 269–287. <https://www.usenix.org/conference/osdi24/presentation/camaioni>
- [8] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC)*. 398–407.
- [9] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)* 43, 2 (1996), 225–267.
- [10] Paulo R Coelho, Nicolas Schiper, and Fernando Pedone. 2017. Fast atomic multicast. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 37–48.
- [11] 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), 1–22.
- [12] James A. Cowling and Barbara Liskov. [n. d.]. Granola: Low-Overhead Distributed Transaction Coordination. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012* (2012), Gernot Heiser and Wilson C. Hsieh (Eds.). USENIX Association, 223–235.
- [13] 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.
- [14] Carole Delporte-Gallet and Hugues Fauconnier. [n. d.]. Fault-Tolerant Genuine Atomic Multicast to Multiple Groups. In *Proceedings of the 4th International Conference on Principles of Distributed Systems, OPODIS 2000, Paris, France, December 20-22, 2000* (2000) (*Studia Informatica Universalis*), Franck Butelle (Ed.). Suger, rue Catulienne Saint-Denis France, 107–122.
- [15] 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 (EuroSys)*. 178–193.
- [16] Udo Fritzke, Philippe Ingels, Achour Mostéfaoui, and Michel Raynal. 1998. Fault-tolerant total order multicast to asynchronous groups. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 228–234.
- [17] 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.
- [18] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, Berlin, Heidelberg, 393–481.
- [19] Rachid Guerraoui and André Schiper. 1997. Genuine Atomic Multicast. In *Distributed Algorithms, 11th International Workshop, WDAG ’97, Saarbrücken, Germany, September 24-26, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1320)*, Marios Mavronicolas and Philippas Tsigas (Eds.). Springer, 141–154. <https://doi.org/10.1007/BFB0030681>
- [20] Rachid Guerraoui and André Schiper. 2001. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science* 254, 1-2 (2001), 297–316.
- [21] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (ATC)*.
- [22] Leslie Lamport. 2005. *Generalized Consensus and Paxos*. Technical Report MSR-TR-2005-33. Microsoft Research.
- [23] Leslie Lamport. 2006. Lower bounds for asynchronous consensus. *Distributed Comput.* 19, 2 (2006), 104–125. <https://doi.org/10.1007/S00446-006-0155-X>
- [24] Butler W Lamson and Howard E Sturgis. 1979. *Crash recovery in a distributed data storage system*. Xerox Palo Alto Research Center Palo Alto, California.
- [25] Iulian Moraru, David G. Andersen, and Michael Kaminsky. [n. d.]. There Is More Consensus in Egalitarian Parliaments. In *Symposium on Operating Systems Principles (SOSP)* (2013).
- [26] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (ATC)*. 305–319.
- [27] Leandro Pacheco, Paulo Coelho, and Fernando Pedone. 2023. PrimCast: A Latency-Efficient Atomic Multicast. In *Proceedings of the 24th International Middleware Conference (Middleware)*. 124–136.
- [28] Seo Jin Park and John Ousterhout. 2019. Exploiting Commutativity For Practical Fast Replication. In *Symposium on Networked Systems Design and Implementation (NSDI)*.
- [29] Fernando Pedone and André Schiper. 1999. Generic broadcast. In *International Symposium on Distributed Computing (PODC)*. Springer, 94–106.
- [30] Fernando Pedone and André Schiper. 2002. Handling message semantics with generic broadcast protocols. *Distributed Computing* 15, 2 (2002), 97–107.
- [31] Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. 2024. SwiftPaxos: Fast Geo-Replicated State Machines. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
- [32] Nicolas Schiper. 2009. *On Multicast Primitives in Large Networks and Partial Replication Protocols*. Ph.D. Dissertation.
- [33] Nicolas Schiper and Fernando Pedone. 2007. *Optimal atomic broadcast and multicast algorithms for wide area networks*. Technical Report. Università della Svizzera italiana.
- [34] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. 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)*. IEEE, 166–175.
- [35] Pierre Sutra. 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 (LIPIcs, Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 35:1–35:19. <https://doi.org/10.4230/LIPIcs.DISC.2022.35>
- [36] Pierre Sutra and Marc Shapiro. 2011. Fast genuine generalized consensus. In *2011 IEEE 30th International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 255–264.
- [37] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. 2017. APUS: fast and scalable paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing (Santa Clara, California) (SoCC)*. Association for Computing Machinery, New York, NY, USA, 94–107. <https://doi.org/10.1145/3127479.3128609>