Solving Atomic Multicast when Groups Crash*

Nicolas Schiper and Fernando Pedone

University of Lugano, Switzerland

Abstract. In this paper, we study the atomic multicast problem, a fundamental abstraction for building fault-tolerant systems. In our model, processes are divided into non-empty and disjoint *groups*. Multicast messages may be addressed to any subset of groups, each message possibly being multicast to a different subset. Several papers previously studied this problem either in local area networks [3, 8, 17] or wide area networks [11, 18]. However, none of them considered atomic multicast when groups may crash. We present two atomic multicast algorithms that tolerate the crash of groups. The first algorithm tolerates an arbitrary number of failures, is genuine (i.e., to deliver a message m, only addressees of m are involved in the protocol), and uses the perfect failures detector $\mathcal{P}$. We show that among realistic failure detectors, i.e., those that do not predict the future, $\mathcal{P}$ is necessary to solve genuine atomic multicast if we do not bound the number of processes that may fail. Thus, $\mathcal{P}$ is the *weakest* realistic failure detector for solving genuine atomic multicast when an arbitrary number of processes may crash. Our second algorithm is non-genuine and less resilient to process failures than the first algorithm but has several advantages: (i) it requires perfect failure detection within groups only, and not across the system, (ii) as we show in the paper it can be modified to rely on unreliable failure detection at the cost of a weaker liveness guarantee, and (iii) it is fast, messages addressed to multiple groups may be delivered within two inter-group message delays only.

1 Introduction

Mission-critical distributed applications typically replicate data in different data centers. These data centers are spread over a large geographical area to provide maximum availability despite natural disasters. Each data center, or *group*, may host a large number of processes connected through a fast local network; a few groups exist, interconnected through high-latency communication links. Application data is replicated locally, for high availability despite the crash of processes in a group, and globally, for locality of access and high availability despite the crash of an entire group.

Atomic multicast is a communication primitive that offers adequate properties, namely agreement on the set of messages delivered and on their delivery order, to implement partial data replication [14, 20]. As opposed to atomic

* The work presented in this paper has been partially funded by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

broadcast [13], atomic multicast allows messages to be addressed to any subset of the groups in the system. For efficiency purposes, multicast protocols should be *genuine* [12], i.e., only the addressees of some message m should participate in the protocol to deliver m. This property rules out the trivial reduction of atomic multicast to atomic broadcast where every message m is broadcast to all groups in the system and only delivered by the addressees of m.

Previous work on atomic multicast [3, 17, 8, 11, 18] all assume that, inside each group, there exists at least one non-faulty process. We here do not make this assumption and allow groups to entirely crash. To the best of our knowledge, this is the first paper to investigate atomic multicast in such a scenario.

The atomic multicast algorithms we present in this paper use oracles that provide possibly inaccurate information about process failures, i.e., failure detectors [5]. Failure detectors are defined by the properties they guarantee on the set of trusted (or suspected) processes they output. Ideally, we would like to find the *weakest* failure detector $\mathcal{D}_{amcast}$ for genuine atomic multicast. Intuitively, $\mathcal{D}_{amcast}$ provides just enough information about process failures to solve genuine atomic multicast but not more. More formally, a failure detector $\mathcal{D}_1$ is at least as strong as a failure detector $\mathcal{D}_2$, denoted as $\mathcal{D}_1 \succeq \mathcal{D}_2$, if and only if there exists an algorithm that implements $\mathcal{D}_2$ using $\mathcal{D}_1$, i.e., the algorithm emulates the output of $\mathcal{D}_2$ using $\mathcal{D}_1$. $\mathcal{D}_{amcast}$ is the weakest failure detector for genuine atomic multicast if two conditions are met: we can use $\mathcal{D}_{amcast}$ to solve genuine atomic multicast (sufficiency) and any failure detector $\mathcal{D}$ that can be used to solve genuine atomic multicast is at least as strong as $\mathcal{D}_{amcast}$, i.e., $\mathcal{D} \succeq \mathcal{D}_{amcast}$ (necessity) [4].

We here consider *realistic* failure detectors only, i.e., those that cannot predict the future [9]. Moreover, we do not assume any bound on the number of processes that can crash. In this context, Delporte *et al.* showed in [9] that the weakest failure detector $\mathcal{D}_{cons}$ for consensus may not make any mistakes about the alive status of processes, i.e., it may not stop trusting a process before it crashes.[1] Additionally, $\mathcal{D}_{cons}$ must eventually stop trusting all crashed processes. In the literature, $\mathcal{D}_{cons}$ is denoted as the perfect failure detector $\mathcal{P}$. Obviously, atomic multicast allows to solve consensus: every process atomically multicasts its proposal; the decision of consensus is the first delivered message. Hence, the weakest realistic failure detector to solve genuine atomic multicast $\mathcal{D}_{amcast}$ when the number of faulty processes is not bounded is at least as strong as $\mathcal{P}$, i.e., $\mathcal{D}_{amcast} \succeq \mathcal{P}$. We show that $\mathcal{P}$ is in fact the weakest realistic failure detector for genuine atomic multicast when an arbitrary number of processes

[1] Intuitively, consensus allows each process to propose a value and guarantees that processes eventually decide on one common value.

may fail by presenting an algorithm that solves the problem using perfect failure detection.

As implementing $\mathcal{P}$ seems hard, if not impossible, in certain settings (e.g., wide area networks), we revisit the problem from a different angle: we consider non-genuine atomic multicast algorithms. For this purpose, as noted above, atomic broadcast could be used. This solution, however, is of little practical interest as delivering messages requires all processes to communicate, even for messages multicast to a single group. The second algorithm we present does not suffer from this problem: messages multicast to a single group g may be delivered without communication between processes outside g. Moreover, our second algorithm offers some advantages when compared to our first algorithm, based on $\mathcal{P}$: Wide-area communication links are used sparingly, messages addressed to multiple groups can be delivered within two inter-group message delays, and perfect failure detection is only required within groups and not across the system. Although this assumption is more reasonable than implementing $\mathcal{P}$ in a wide area network, it may still be too strong for some systems. Thus, we discuss a modification to the algorithm that tolerates unreliable failure detection, at the cost of a weaker liveness guarantee. The price to pay for the valuable features of this second algorithm is a lower process failure resiliency: group crashes are still tolerated provided that *enough* processes in the whole system are correct.

Contribution In this paper, we make the following contributions. We present two atomic multicast algorithms that tolerate group crashes. The first algorithm is genuine, tolerates an arbitrary number of failures, and requires perfect failure detection. The second algorithm is non-genuine but only requires perfect failure detection inside each group and may deliver messages addressed to multiple groups in two inter-group message delays. We present a modification to the algorithm to cope with unreliable failure detection.

Road map The rest of the paper is structured as follows. Section 2 reviews the related work. In Section 3 our system model and definitions are introduced. Sections 4 and 5 present the two atomic multicast algorithms. Finally, Section 6 concludes the paper. The proof of correctness of the algorithms can be found in [19].

2 Related Work

The literature on atomic broadcast and multicast algorithms is abundant [7]. We briefly review some of the relevant papers on atomic multicast.

In [12], the authors show the impossibility of solving genuine atomic multicast with unreliable failure detectors when groups are allowed to intersect. Hence, the algorithms cited below consider non-intersecting groups. Moreover, they all assume that groups do not crash, i.e., there exists at least one correct process inside each group.

These algorithms can be viewed as variations of Skeen's algorithm [3], a multicast algorithm designed for failure-free systems, where messages are associated with timestamps and the message delivery follows the timestamp order. In [17], the addressees of a message m, i.e., the processes to which m is multicast, exchange the timestamp they assigned to m, and, once they receive this timestamp from a majority of processes of each group, they propose the maximum value received to consensus. Because consensus is run among the addressees of a message and can thus span multiple groups, this algorithm is not well-suited for wide area networks. In [8], consensus is run inside groups exclusively. Consider a message m that is multicast to groups $g_1, ..., g_k$. The first destination group of m, g_1, runs consensus to define the final timestamp of m and hands over this message to group g_2. Every subsequent group proceeds similarly up to g_k. To ensure agreement on the message delivery order, before handling other messages, every group waits for a final acknowledgment from group g_k. In [11], inside each group g, processes implement a logical *clock* that is used to generate timestamps, this is g's clock (consensus is used among processes in g to maintain g's clock). Every multicast message m goes through four stages. In the first stage, in every group g addressed by m, processes define a timestamp for m using g's clock. This is g's proposal for m's final timestamp. Groups then exchange their proposals and set m's final timestamp to the maximum among all proposals. In the last two stages, the clock of g is updated to a value bigger than m's final timestamp and m is delivered when its timestamp is the smallest among all messages that are in one of the four stages. In [18], the authors present an optimization of [11] that allows messages to skip the second and third stages in certain conditions, therefore sparing the execution of consensus instances. The algorithms of [11, 18] can deliver messages in two inter-group message delays; [18] shows that this is optimal.

To the best of our knowledge, this is the first paper that investigates the solvability of atomic multicast when groups may entirely crash. Two algorithms are presented: the first one is genuine but requires system-wide perfect failure detection. The second algorithms is not genuine but only requires perfect failure detection inside groups.

3 Problem Definition

3.1 System Model

We consider a system $\Pi = \{p_1, ..., p_n\}$ of processes which communicate through message passing and do not have access to a shared memory or a global clock. Processes may however access failure detectors [5]. We assume the benign crash-stop failure model: processes may fail by crashing, but do not behave maliciously. A process that never crashes is $correct$; otherwise it is $faulty$. The maximum number of processes that may crash is denoted by f. The system is asynchronous, i.e., messages may experience arbitrarily large (but finite) delays and there is no bound on relative process speeds. Furthermore, the communication links do not corrupt nor duplicate messages, and are quasi-reliable: if a correct process p sends a message m to a correct process q, then q eventually receives m. We define $\Gamma = \{g_1, ..., g_m\}$ as the set of process groups in the system. Groups are disjoint, non-empty and satisfy $\bigcup_{g \in \Gamma} g = \Pi$. For each process $p \in \Pi$, $group(p)$ identifies the group p belongs to. A group g that contains at least one correct process is correct; otherwise g is faulty.

3.2 Atomic Multicast

Atomic multicast allows messages to be A-MCast to any subset of groups in Γ. For every message m, $m.dst$ denotes the groups to which m is multicast. Let p be a process. By abuse of notation, we write $p \in m.dst$ instead of $\exists g \in \Gamma : g \in m.dst \wedge p \in g$. Atomic multicast is defined by the primitives A-MCast and A-Deliver and satisfies the following properties: (i) *uniform integrity:* For any process p and any message m, p A-Delivers m at most once, and only if $p \in m.dst$ and m was previously A-MCast, (ii) *validity:* if a correct process p A-MCasts a message m, then eventually all correct processes $q \in m.dst$ A-Deliver m, (iii) *uniform agreement:* if a process p A-Delivers a message m, then all correct processes $q \in m.dst$ eventually A-Deliver m, and (iv) *uniform prefix order:* for any two messages m and m' and any two processes p and q such that $\{p, q\} \in m.dst \cap m'.dst$, if p A-Delivers m and q A-Delivers m', then either p A-Delivers m' before m or q A-Delivers m before m'.

Let $\mathcal{A}$ be an algorithm solving atomic multicast. We define $\mathcal{R}(\mathcal{A})$ as the set of all admissible runs of $\mathcal{A}$. We require atomic multicast algorithms to be *genuine* [12]:

- *Genuineness*: An algorithm $\mathcal{A}$ solving atomic multicast is said to be *genuine* iff for any run $R \in \mathcal{R}(\mathcal{A})$ and for any process p, in R, if p sends or receives a message then some message m is A-MCast and either p is the process that A-MCasts m or $p \in m.dst$.

4 Solving Atomic Multicast with a Perfect Failure Detector

In this section, we present the first genuine atomic multicast algorithm that tolerates an arbitrary number of process failures, i.e., $f \leq n$. We first define additional abstractions used in the algorithm, then explain the mechanisms to ensure agreement on the delivery order, and finally, we present the algorithm itself.

4.1 Additional Definitions and Assumptions

Failure Detector $\mathcal{P}$ We assume that processes have access to the perfect failure detector $\mathcal{P}$ [5]. This failure detector outputs a list of trusted processes and satisfies the following properties[2]: (i) *strong completeness:* eventually no faulty process is ever trusted by any correct process and (ii) *strong accuracy:* no process stops being trusted before it crashes.

Causal Multicast The algorithm we present below uses a *causal multicast* abstraction. Causal multicast is defined by primitives ***C-MCast(m)*** and ***C-Deliver(m)***, and satisfies the uniform integrity, validity, and uniform agreement properties of atomic multicast as well as the following *uniform causal order* property: for any messages m and m', if C-MCast(m) $\rightarrow$ C-MCast(m'), then no process $p \in m.dst \cap m'.dst$ C-Delivers m' unless it has previously C-Delivered m.[3] To the best of our knowledge, no algorithm implementing this specification of causal multicast exists. We thus present a genuine causal multicast algorithm that tolerates an arbitrary number of failures in [19].[4]

Global Data Computation We also assume the existence of a *global data computation* abstraction [10]. The global data computation problem consists in providing each process with the same vector V, with one entry per process, such that each entry is filled with a value provided by the corresponding process. Global data computation is defined by the primitives propose(v) and decide(V) and satisfies the following properties: (i) *uniform validity:* if a process p decides V, then $\forall q : V[q] \in \{v_q, \perp\}$, where v_q is q's proposal, (ii) *termination:* if every correct process proposes a value, then every correct process eventually decides one vector, (iii) *uniform agreement:* if a process p decides V, then all correct

[2] Historically, $\mathcal{P}$ was defined to output a set of suspected processes. We here define its output as a set of trusted processes, i.e., in our definition the output corresponds to the complement of the output in the original definition.

[3] The relation $\rightarrow$ is Lamport's transitive happened before relation on events [15]. Here, events can be of two types, C-MCast or C-Deliver. The relation is defined as follows: $e_1 \rightarrow e_2 \Leftrightarrow$ e_1, e_2 are two events on the same process and e_1 happens before e_2 or $e_1 =$ C-MCast(m) and $e_2 =$ C-Deliver(m) for some message m.

[4] The genuineness of causal multicast is defined in a similar way as for atomic multicast.

processes q eventually decide V, and (iv) *uniform obligation:* if a process p decides V, then $V[p] = v_p$. An algorithm that solves global data computation using the perfect failure detector $\mathcal{P}$ appears in [10]. This algorithm tolerates an arbitrary number of failures.

4.2 Agreeing on the Delivery Order

The algorithm associates every multicast message with a timestamp. To guarantee agreement on the message delivery order, two properties are ensured: (1) processes agree on the message timestamps and (2) after a process p A-Delivers a message with timestamp ts, p does not A-Deliver a message with a smaller timestamp than ts. These properties are implemented as described next.

For simplicity, we initially assume a multicast primitive that guarantees agreement on the set of messages processes deliver, but not causal order; we then show how this algorithm may incur into problems, which can be solved using causal multicast. To A-MCast a message m_1, m_1 is thus first multicast to the addressees of m_1. Upon delivery of m_1, every process p uses a local variable, denoted as TS_p, to define its proposal for m_1's timestamp, $m_1.ts_p$. Process p then proposes $m_1.ts_p$ in m_1's global data computation (gdc) instance. The definitive timestamp of m_1, $m_1.ts^{def}$, is the maximum value of the decided vector V. Finally, p sets TS_p to a bigger value than $m_1.ts^{def}$ and A-Delivers m_1 when all *pending* messages have a bigger timestamp than $m_1.ts^{def}$—a message m is pending if p delivered m but did not A-Deliver m yet.

Although this reasoning ensures that processes agree on the message delivery order, the delivery sequence of faulty processes may contain *holes*. For instance, p may A-Deliver m_1 followed by m_2, while some faulty process q only A-Delivers m_2. To see why, consider the following scenario. Process p delivers m_1 and m_2, and proposes some timestamp ts_p for these two messages. As q is faulty, it may only deliver m_2 and propose some timestamp ts_q bigger than ts_p as m_2's timestamp—this is possible because q may have A-Delivered several messages before m_2 that were not addressed to p and q thus updated its TS variable. Right after deciding in m_2's gdc instance, q A-Delivers m_2 and crashes. Later, p decides in m_1 and m_2's gdc instances, and A-Delivers m_1 followed by m_2, as m_1's definitive timestamp is smaller than m_2's.

To solve this problem, before A-Delivering a message m, every process p addressed by m computes m's *potential predecessor set*, denoted as $m.pps$. This set contains all messages addressed to p that may potentially have a smaller definitive timestamp than m's (in the example above, m_1 belongs to $m_2.pps$).[5]

[5] Note that the idea of computing a message's potential predecessor set appears in the atomic multicast algorithm of [17]. However, this algorithm assumes a majority of correct processes in every group and thus computes this set differently.

Message m is then A-Delivered when for all messages m' in $m.pps$ either (a) $m'.ts^{def}$ is known and it is bigger than $m.ts^{def}$ or (b) m' has been A-Delivered already.

The potential predecessor set of m is computed using causal multicast: To A-MCast m, m is first causally multicast. Second, after p decides in m's instance and updates its TS variable, p causally multicasts an *ack* message to the destination processes of m. As soon as p receives an *ack* message from all processes addressed by m that are trusted by its perfect failure detector module, the potential predecessor set of m is simply the set of pending messages.

Intuitively, m's potential predecessor set is correctly constructed for the two following facts: (1) Any message m', addressed to p and some process q, that q causally delivers *before* multicasting m's *ack* message will be in $m.pps$ (the definitive timestamp of m' might be smaller than m's). (2) Any message causally delivered by some addressee q of m *after* multicasting m's *ack* message will have a bigger definitive timestamp than m's. Fact (1) holds from causal order, i.e., if q C-Delivers m' before multicasting m's *ack* message, then p C-Delivers m' before C-Delivering m's *ack*. Fact (2) is a consequence of the following. As p's failure detector module is perfect, p stops waiting for *ack* messages as soon as p received an *ack* from all *alive* addressees of m. Hence, since processes update their TS variable after deciding in m's global data computation instance but before multicasting the *ack* message of m, no addressee of m proposes a timestamp smaller than $m.ts^{def}$ *after* multicasting m's *ack* message.

4.3 The Algorithm

Algorithm $\mathcal{A}1$ is composed of four tasks. Each line of the algorithm, task 2, and the procedure ADeliveryTest are executed atomically. Messages are composed of application data plus four fields: dst, id, ts, and $stage$. For every message m, $m.dst$ indicates to which groups m is A-MCast, $m.id$ is m's unique identifier, $m.ts$ denotes m's current timestamp, and $m.stage$ defines in which stage m is. We explain Algorithm $\mathcal{A}1$ by describing the actions a process p takes when a message m is in one of the three possible stages: s_0, s_1, or s_2.

To A-MCast m, m is first C-MCast to its addressees (line 8). In stage s_0, p C-Delivers m, sets m's timestamp proposal, and adds m to the set of pending messages $Pending$ (lines 10-12). In stage s_1, p computes $m.ts^{def}$ (lines 17-19) and ensures that all messages in $m.pps$ are in p's pending set (lines 20-23), as explained above. Finally, in stage s_2, m is A-Delivered when for all messages m' in $m.pps$ that are still in p's pending set (if m' is not in p's pending set anymore, m' was A-Delivered before), m' is in stage s_2 (and thus $m'.ts$ is the definitive timestamp of m') and $m'.ts$ is bigger than $m.ts$ (lines 4-6). Notice that if m and m' have the same timestamp, we break ties using their

message identifiers. More precisely, $(m.ts, m.id) < (m'.ts, m'.id)$ holds if either $m.ts < m'.ts$ or $m.ts = m'.ts$ and $m.id < m'.id$. Figure 1 illustrates a failure-free run of the algorithm.

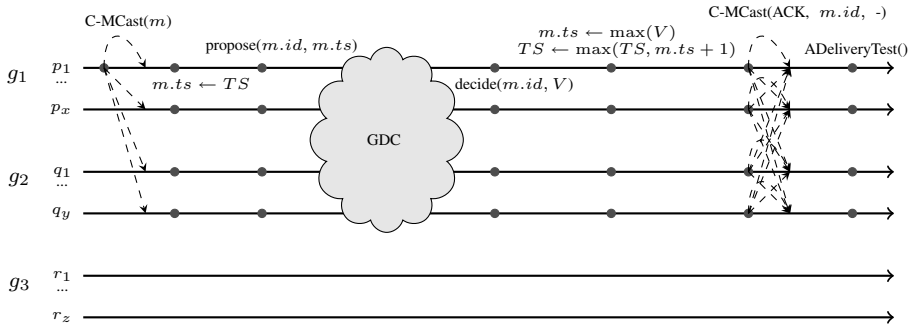

Fig. 1. Algorithm $\mathcal{A}1$ in the failure-free case when a message m is A-MCast to groups g_1 and g_2.

5 Solving Atomic Multicast with Weaker Failure Detectors

In this section, we solve atomic multicast with a non-genuine algorithm. The Algorithm $\mathcal{A}2$ we present next does not require system-wide perfect failure detection and delivers messages in fewer communication steps. We first define additional abstractions used by the algorithm and summarize its assumptions. We then present the algorithm itself and conclude with a discussion on how to further reduce its delivery latency and weaken its failure detection requirements.

5.1 Additional Definitions and Assumptions

Failure Detector $\diamond\mathcal{P}$ We assume that processes have access to an eventually perfect failure detector $\diamond\mathcal{P}$ [5]. This failure detector ensures the strong completeness property of $\mathcal{P}$ and the following *eventual strong accuracy* property: there is a time after which no process stops being trusted before it crashes.

Reliable Multicast Reliable multicast is defined by the primitives R-MCast and R-Deliver and ensures all properties of causal multicast except uniform causal order.

Algorithm $\mathcal{A}1$ Genuine Atomic Multicast using $\mathcal{P}$ - Code of process p

```
1: Initialization
2:    TS ← 1, Pending ← ∅

3: procedure ADeliveryTest()
4:    while ∃m ∈ Pending : m.stage = s2
            ∀id ∈ m.pps : ∃m' ∈ Pending : m'.id = id ⇒
                          m'.stage = s2 ∧ (m.ts, m.id) < (m'.ts, m'.id) do
5:       A-Deliver(m)
6:       Pending ← Pending \ {m}

7: To A-MCast message m                                              {Task 1}
8:    C-MCast m to m.dst

9: When C-Deliver(m) atomically do                                   {Task 2}
10:   m.ts ← TS
11:   m.stage ← s0
12:   Pending ← Pending ∪ {m}

13: When ∃m ∈ Pending : m.stage = s0                                 {Task 3}
14:   m.stage ← s1
15:   fork task ConsensusTask(m)

16: ConsensusTask(m)                                                 {Task x}
17:   Propose(m.id, m.ts)                          ▷ GDC among processes in m.dst
18:   wait until Decide(m.id, V)
19:   m.ts ← max(V)
20:   TS ← max(TS, m.ts + 1)
21:   C-MCast(ACK, m.id, p) to m.dst
22:   wait until ∀q ∈ P ∩ m.dst : C-Deliver(ACK, m.id, q)
23:   m.pps ← {m'.id | m' ∈ Pending ∧ m' ≠ m}
24:   m.stage ← s2
25:   atomic block
26:      ADeliveryTest()
```

Consensus In the *consensus* problem, processes propose values and must reach agreement on the value decided. Consensus is defined by the primitives propose(v) and decide(v) and satisfies the following properties [13]: (i) *uniform validity:* if a process decides v, then v was previously proposed by some process, (ii) *termination:* if every correct process proposes a value, then every correct process eventually decides exactly one value, and (iii) *uniform agreement:* if a process decides v, then all correct processes eventually decide v.

Generic Broadcast Generic broadcast ensures the same properties as atomic multicast except that all messages are addressed to all groups and only *conflicting* messages are totally ordered. More precisely, generic broadcast ensures uniform integrity, validity, uniform agreement, and the following *uniform generalized order* property: for any two conflicting messages m and m' and any

two processes p and q, if p G-Delivers m and q G-Delivers m', then either p G-Delivers m' before m or q G-Delivers m before m'.

Assumptions To solve generic broadcast, either a simple majority of correct processes must be correct, i.e., $f < n/2$, and non-conflicting messages may be delivered in three message delays [2] or a two-third majority of processes must be correct, i.e., $f < n/3$, and non-conflicting message may be delivered in two message delays [16]. Both algorithms require a system-wide leader failure detector Ω [4], and thus the eventual perfect failure detector $\diamond\mathcal{P}$ we assume is sufficient. Moreover, inside each group, we need consensus and reliable multicast abstractions that tolerate an arbitrary number of failures. For this purpose, among realistic failure detectors, $\mathcal{P}$ is necessary and sufficient for consensus [9] and sufficient for reliable multicast [1].[6] Note that in practice, implementing $\mathcal{P}$ within each group is more reasonable than across the system, especially if groups are inside local area networks. We discuss below how to remove this assumption.

5.2 Algorithm Overview

The algorithm is inspired by the atomic broadcast algorithm of [18]. We first recall its main ideas and then explain how we cope with group failures—[18] assumes that there is at least one correct process in every group. We then show how *local* messages to some group g, i.e., messages multicast from processes inside g and addressed to g only, may be delivered with no inter-group communication at all.

To A-MCast a message m, a process p R-MCasts m to p's group. In parallel, processes execute an *unbounded* sequence of rounds. At the end of each round, processes A-Deliver a set of messages according to some deterministic order. To ensure agreement on the messages A-Delivered in round r, processes proceed in two steps. In the first step, inside each group g, processes use consensus to define g's bundle of messages. In the second step, groups exchange their message bundles. The set of message A-Delivered by some process p at the end of round r is the union of all bundles, restricted to messages addressed to p.

In case of group crashes, this solution does not ensure liveness however. Indeed, if a group g crashes there will be some round r after which no process receives the message bundles of g. To circumvent this problem we proceed in two steps: (a) we allow processes to stop waiting for g's message bundle, and (b) we let processes agree on the set of message bundles to consider for each round.

[6] In [1], the authors present the weakest failure detector to solve reliable broadcast. Extending the algorithm of [1] to the multicast case using the same failure detector is straightforward.

To implement (a), processes maintain a common *view* of the groups that are trusted to be alive, i.e., groups that contain at least one alive process. Processes then wait for the message bundles from the groups currently in the view. A group g may be erroneously removed from the view, if it was mistakenly suspected of having crashed. Therefore, to ensure that message m multicast by a correct process will be delivered by all correct addressees of m, we allow members of g to add their group back to the view. To achieve (b), processes agree on the sequence of views and the set of message bundles between each view change. For this purpose, we use a generic broadcast abstraction to propagate message bundles and view change messages, i.e., messages to add or remove groups. Since message bundles can be delivered in different orders at different processes, provided that they are delivered between the same two view change messages, we define the message conflict relation as follows: view change messages conflict with all messages and message bundles only conflict with view change messages. As view change messages are not expected to be broadcast often, such a conflict relation definition allows for faster message bundle delivery.

Processes may also A-Deliver local messages to some group g without communicating with processes outside of g. As these messages are addressed to g only, members of g may A-Deliver them directly after consensus, and thus before receiving the groups' message bundles.

We note that maintaining a common view of the alive groups in the system resembles what is called in the literature group membership [6]. Intuitively, a group membership service provides processes with a consistent view of alive processes in the system, i.e., processes "see" the same sequence of views. Moreover, processes agree on the set of messages delivered between each view change, a property that is required for message bundles.[7] In fact, our algorithm could have been built on top of such an abstraction. However, doing so would have given us less freedom to optimize the delivery latency of message bundles.

5.3 The Algorithm

Algorithm $\mathcal{A}2$ is composed of five tasks. Each line of the algorithm is executed atomically. On every process p, six global variables are used:Rnd denotes the current round number, $Rdelivered$ and $Adelivered$ are the set of R-Delivered and A-Delivered messages respectively, $Gdelivered$ is the sequence of G-Delivered messages, $MsgBundle$ stores the message bundles, and $View$ is the set of groups currently deemed to be alive.

In the algorithm, every G-BCast message m has the following format: $(rnd, g, type, msgs)$, where rnd denotes the round in which m was G-BCast,

[7] Some group membership specifications also guarantee total ordering of the messages delivered between view changes.

g is the group m refers to, $type$ denotes m's type and is either $msgBundle$, add, or $remove$, and $msgs$ is a set of messages; this field is only used if m is a message bundle.

To A-MCast a message m, a process p R-MCasts m to p's group (line 5). In every round r, the set of messages that have been R-Delivered but not A-Delivered yet are proposed to the next consensus instance (line 9), p A-Delivers the set of local messages decided in this instance (line 12), and global messages, i.e., non local messages, are G-BCast at line 15 if $group(p)$ belongs to the view. Otherwise, p G-BCasts a message to add $group(p)$ to the view.

Process p then gathers message bundles of the current round k using variable $MsgBundle$: Process p executes the while loop of lines 19-26 until, for every group g, $MsgBundle[g]$ is neither $\perp$, i.e. p is not waiting to receive a message bundle from g, nor $\top$, a value whose signification is explained below. The first message m_g^k of round k related to g of type $msgBundle$ or $remove$ that p G-Delivers "locks" $MsgBundle[g]$, i.e., any subsequent G-Delivered message of round k concerning g is discarded (line 23). If m_g^k is of type $msgBundle$, p stores g's message bundle in $MsgBundle[g]$ (line 26). Otherwise, m_g^k was G-BCast by some process q that suspected g to have entirely crashed, i.e., failure detector $\diamond\mathcal{P}$ at q did not trust any member of g (lines 33-35), and thus p sets $MsgBundle[g]$ to $\emptyset$ (line 25). Note that q sets $MsgBundle[g]$ to $\top$ after G-BCasting a message of the form $(k, g, remove, \text{-})$ to prevent q from G-BCasting multiple "remove g" messages in the same round.

While p is gathering message bundles for round k, it may also handle some message of type add concerning g, in which case p adds g to a local variable $groupsToAdd$ (line 24). Note that this type of message is not tagged with a round number to ensure that messages A-MCast from correct groups are eventually A-Delivered by their correct addressees. In fact, tagging add messages with the round number could prevent a group from being added to the view as we now explain. Consider a correct group g that is removed from the view in the first round. In every round, members of g G-BCast a message to add g back to the view. In every round however, processes G-Deliver message bundles of groups in the view before G-Delivering these "add g" messages, and they are thus discarded.

After exiting from the while loop, p A-Delivers global messages (line 28), the view is recomputed as the groups g such that $MsgBundle[g] \neq \emptyset$ or $g \in groupsToAdd$ (line 30), and p sets $MsgBundle[g]$ to either $\perp$, if g belongs to the new view, or $\emptyset$ otherwise (p will not wait for a message bundle from g in the next round). Figures 2 and 3 respectively illustrate a failure-free run of the algorithm and a run where group g_3 entirely crashes.

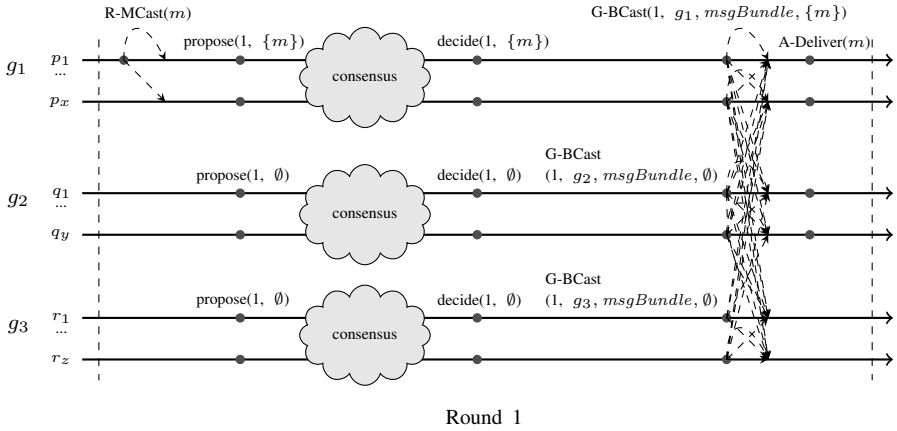

Fig. 2. Algorithm $\mathcal{A}2$ in the failure-free case when a message m is A-MCast to groups g_1 and g_2.

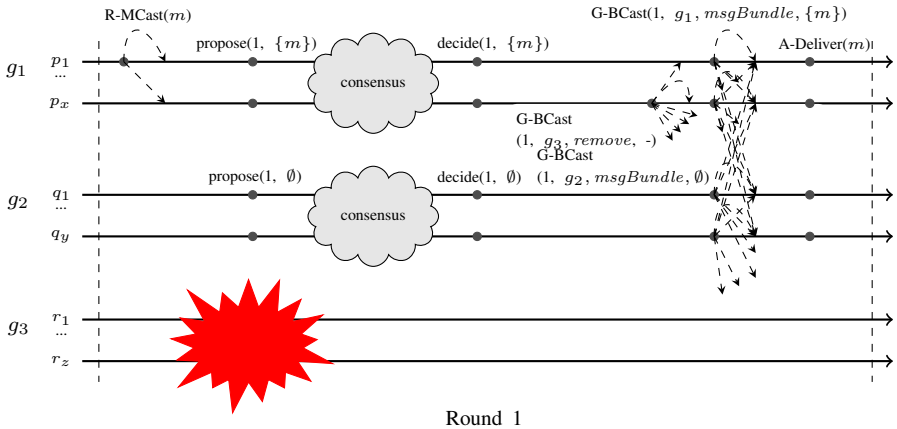

Fig. 3. Algorithm $\mathcal{A}2$ when group g_3 crashes and a message m is A-MCast to groups g_1 and g_2.

5.4 Further Improvements

Delivery Latency In Algorithm $\mathcal{A}2$, local messages are delivered directly after consensus. Hence, these messages do not bear the cost of a single inter-group message delay unless: (a) they are multicast from a group different than their destination group or (b) they are multicast while the groups' bundle of messages are being exchanged, in which case the next consensus instance can only

be started when message bundles of the current round have been received. Obviously, nothing can be done to avoid case (a). However, we can prevent case

Algorithm $\mathcal{A}2$ Non-Genuine Atomic Multicast - Code of process p

1: **Initialization**
2: $Rnd \leftarrow 1$, $Rdelivered \leftarrow \emptyset$, $Adelivered \leftarrow \emptyset$, $Gdelivered \leftarrow \epsilon$
3: $View \leftarrow \Gamma$, $MsgBundle[g] \leftarrow \perp$ for each group $g \in \Gamma$

4: **To A-MCast** message m {*Task 1*}
5: R-MCast m to $group(p)$

6: **When** R-Deliver(m) {*Task 2*}
7: $Rdelivered \leftarrow Rdelivered \cup \{m\}$

8: **Loop** {*Task 3*}
9: Propose(Rnd, $Rdelivered \setminus Adelivered$) ▷ consensus inside group
10: **wait until** Decide(Rnd, $msgs$)

11: $localMsgs \leftarrow \{m \mid m \in msgs \ \wedge \ m.dst = \{group(p)\}\}$
12: **A-Deliver** messages in $localMsgs$ in some deterministic order
13: $Adelivered \leftarrow Adelivered \cup localMsgs$

14: **if** $group(p) \in View$ **then**
15: G-BCast($Rnd, group(p), msgBundle, msgs \setminus localMsgs$)
16: **else**
17: G-BCast(-, $group(p)$, add, -)
18: $groupsToAdd \leftarrow \emptyset$

19: **while** $\exists g \in \Gamma : MsgBundle[g] \in \{\perp, \top\}$
20: **if** $\nexists (rnd, g, type, msgs) \in Gdelivered : (rnd = Rnd \ \vee \ type = add)$ **then**
21: **wait until** G-Deliver($rnd, g, type, msgs$) $\wedge$ $(rnd = Rnd \ \vee \ type = add)$
22: $(rnd, g', type, msgs) \leftarrow$ remove first message in $Gdelivered$ s.t. $(rnd = Rnd \ \vee \ type = add)$
23: **if** $MsgBundle[g'] \in \{\perp, \top\}$ **then**
24: **if** $type = add$ **then** $groupsToAdd \leftarrow groupsToAdd \cup \{g'\}$
25: **else if** $type = remove$ **then** $MsgBundle[g'] \leftarrow \emptyset$
26: **else** $MsgBundle[g'] \leftarrow msgs$
27: $globalMsgs \leftarrow \{m \mid \exists g \in \Gamma : MsgBundle[g] = msgs \ \wedge \ m \in msgs\}$
28: **A-Deliver** messages in $globalMsgs$ addressed to p in some deterministic order
29: $Adelivered \leftarrow Adelivered \cup globalMsgs$

30: $View \leftarrow \{g \mid MsgBundle[g] \neq \emptyset\} \cup groupsToAdd$
31: **foreach** $g \in \Gamma : MsgBundle[g] \leftarrow \perp$ (if $g \in View$) or $\emptyset$ (otherwise)
32: $Rnd \leftarrow Rnd + 1$

33: **When** $\exists g \in View : MsgBundle[g] = \perp \ \wedge \ \forall q \in g : q \notin \Diamond\mathcal{P}$ {*Task 4*}
34: G-BCast($Rnd, g, remove$, -)
35: $MsgBundle[g] \leftarrow \top$

36: **When** G-Deliver($type, m$) {*Task 5*}
37: $Gdelivered \leftarrow Gdelivered \oplus (rnd, g, type, msgs)$

(b) from happening by allowing rounds to overlap. That is, we start the next round before receiving the groups' bundle of messages for the current round. Note that to ensure agreement on the relative delivery order of local and global messages, processes inside the same group must agree on when global messages of a given round are delivered, i.e., after which consensus instance. For this purpose, a mapping between rounds and consensus instances can be defined. To control the inter-group traffic, we may also specify that message bundles are sent, say every κ consensus instance. Choosing κ presents a trade-off between inter-group traffic and delivery latency of global messages.

Failure Detection To weaken the failure detector required inside each group, i.e., $\mathcal{P}$ in Algorithm $\mathcal{A}2$, we may remove a group g from the view as soon as a majority of processes in g are suspected. This allows to use consensus and reliable multicast algorithms that are safe under an arbitrary number of failures and live only when a majority of processes are correct. Hence, the leader failure detector Ω becomes sufficient. Care should be taken as when to add g to the view again: this should only be done when a majority of processes in g are trusted to be alive. This solution ensures a weaker liveness guarantee however: correct processes in some group g will *successfully* multicast and deliver messages only if g is *maj-correct*, i.e., g contains a majority of correct processes. More precisely, the liveness guaranteed by this modified algorithm is as follows (uniform integrity and uniform prefix order remain unchanged):

- *weak uniform agreement:* if a process p A-Delivers a message m, then all correct processes $q \in m.dst$ in a maj-correct group eventually A-Deliver m
- *weak validity:* if a correct process p in a maj-correct group A-MCasts a message m, then all correct processes $q \in m.dst$ in a maj-correct group eventually A-Deliver m.

6 Final Remarks

In this paper, we addressed the problem of solving atomic multicast in the case where groups may entirely crash. We presented two algorithms. The first algorithm is genuine, tolerates an arbitrary number of process failures, and requires perfect failure detection. We showed, in Section 1, that if we consider realistic failure detectors only and we do not bound the number of failures, $\mathcal{P}$ is necessary to solve this problem. The second algorithm we presented is not genuine but requires perfect failure detection inside each group only and may deliver messages addressed to multiple groups within two inter-group message delays. We showed how this latter algorithm can be modified to cope with unreliable failure detection, at the cost of a weaker liveness guarantee.

Figure 4 provides a comparison of the presented algorithms with the related work. The best-case message delivery latency is computed by considering a message A-MCast to k groups ($k \geq 2$) in a failure-free scenario when the inter-group message delay is δ and the intra-group message delay is negligible. Note that we took 2δ as the best-case latency for causal multicast [19] and global data computation [10].

Algorithm	genuine?	resiliency	failure detector(s)	best-case latency
[8]	yes	majority correct in each group	group-wide Ω	$(k+1)\delta$
[17]	yes	majority correct in each group	group-wide Ω	4δ
[11]	yes	majority correct in each group	group-wide Ω	2δ
[18]	yes	majority correct in each group	group-wide Ω	2δ
$\mathcal{A}1$	yes	$f \leq n$	system-wide $\mathcal{P}$	6δ
$\mathcal{A}2$	no	$f < n/2$	group-wide $\mathcal{P}$ and system-wide $\diamond\mathcal{P}$ (modification of algorithm with weaker liveness tolerates unreliable failure detection)	3δ
		$f < n/3$		2δ

Fig. 4. Comparison of the presented algorithms and related work.

References

1. M. K. Aguilera, S. Toueg, and B. Deianov. Revising the weakest failure detector for uniform reliable broadcast. In *Proceedings of DISC'99*, pages 19–33. Springer-Verlag, 1999.
2. M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *Proceedings of DISC'00*, pages 268–283. Springer-Verlag, 2000.
3. K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.
4. T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.
5. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.
6. G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.
7. X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.
8. C. Delporte-Gallet and H. Fauconnier. Fault-tolerant genuine atomic multicast to multiple groups. In *Proceedings of OPODIS'00*, pages 107–122. Suger, Saint-Denis, rue Catulienne, France, 2000.
9. C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. A realistic look at failure detectors. In *Proceedings of DSN'02*, pages 345–353. IEEE Computer Society, 2002.
10. C. Delporte-Gallet, H. Fauconnier, J.-M. Helary, and M. Raynal. Early stopping in global data computation. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):909–921, 2003.

11. U. Fritzke, Ph. Ingels, A. Mostéfaoui, and M. Raynal. Fault-tolerant total order multicast to asynchronous groups. In *Proceedings of SRDS'98*, pages 578–585. IEEE Computer Society, 1998.
12. R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 254(1-2):297–316, 2001.
13. V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 1993.
14. Udo Fritzke Jr. and Philippe Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *Proceedings of ICDCS'01*, pages 284–291. IEEE Computer Society, 2001.
15. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
16. F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.
17. L. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *Proceedings of IC3N'98*. IEEE, 1998.
18. N. Schiper and F. Pedone. On the inherent cost of atomic broadcast and multicast in wide area networks. In *Proceedings of ICDCN'08*, pages 147–157. Springer, 2008.
19. N. Schiper and F. Pedone. Solving atomic multicast when groups crash. Technical Report 2008/002, University of Lugano, 2008.
20. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.