The Weakest Failure Detector for Genuine Atomic Multicast

Pierre Sutra
Telecom SudParis, Palaiseau, France

Abstract

Atomic broadcast is a group communication primitive to order messages across a set of distributed processes. Atomic multicast is its natural generalization where each message m is addressed to $dst(m)$, a subset of the processes called its destination group. A solution to atomic multicast is *genuine* when a process takes steps only if a message is addressed to it. Genuine solutions are the ones used in practice because they have better performance.

Let $\mathcal{G}$ be all the destination groups and $\mathcal{F}$ be the cyclic families in it, that is the subsets of $\mathcal{G}$ whose intersection graph is hamiltonian. This paper establishes that the weakest failure detector to solve genuine atomic multicast is $\mu = (\wedge_{g,h \in \mathcal{G}}\ \Sigma_{g \cap h}) \wedge (\wedge_{g \in \mathcal{G}}\ \Omega_g) \wedge \gamma$, where Σ_P and Ω_P are the quorum and leader failure detectors restricted to the processes in P, and γ is a new failure detector that informs the processes in a cyclic family $\mathfrak{f} \in \mathcal{F}$ when $\mathfrak{f}$ is faulty.

We also study two classical variations of atomic multicast. The first variation requires that message delivery follows the real-time order. In this case, μ must be strengthened with $1^{g \cap h}$, the indicator failure detector that informs each process in $g \cup h$ when $g \cap h$ is faulty. The second variation requires a message to be delivered when the destination group runs in isolation. We prove that its weakest failure detector is at least $\mu \wedge (\wedge_{g,h \in \mathcal{G}}\ \Omega_{g \cap h})$. This value is attained when $\mathcal{F} = \varnothing$.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Software and its engineering → Distributed systems organizing principles; General and reference → Reliability

Keywords and phrases Failure Detector, State Machine Replication, Consensus

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.35

Related Version *Extended Version*: `https://arxiv.org/abs/2208.07650` [37]

1 Introduction

Context. Multicast is a fundamental group communication primitive used in modern computing infrastructures. This primitive allows to disseminate a message to a subset of the processes in the system, its destination group. Implementations exist over point-to-point protocols such as the Internet Protocol. Multicast is atomic when it offers the properties of atomic broadcast to the multicast primitive: each message is delivered at most once, and delivery occurs following some global order. Atomic multicast is used to implement strongly consistent data storage [4, 11, 36, 32].

It is easy to see that atomic multicast can be implemented atop atomic broadcast. Each message is sent through atomic broadcast and delivered where appropriate. Such a naive approach is however used rarely in practice because it is inefficient when the number of destination groups is large [31, 35]. To rule out naive implementations, Guerraoui and Schiper [25] introduce the notion of genuineness. An implementation of atomic multicast is *genuine* when a process takes steps only if a message is addressed to it.

Existing genuine atomic multicast algorithms that are fault-tolerant have strong synchrony assumptions on the underlying system. Some protocols (such as [34]) assume that a perfect failure detector is available. Alternatively, a common assumption is that the destination groups are decomposable into disjoint groups, each of these behaving as a logically correct entity. Such an assumption is a consequence of the impossibility result established in [25].

Table 1 About the weakest failure detector for atomic multicast. *($\sqrt{\sqrt{}}$= strongly genuine)*

Genuiness	Order	Weakest	
$\times$	Global	$\Omega \wedge \Sigma$	[8, 15]
$\surd$	·	$\notin \mathcal{U}_2$	[25]
·	·	$\leq P$	[34]
·	·	μ	§5, §4
·	Strict	$\mu \wedge (\wedge_{g,h \in \mathcal{G}}\ 1^{g \cap h})$	§6.1
·	Pairwise	$(\wedge_{g,h \in \mathcal{G}}\ \Sigma_{g \cap h}) \wedge (\wedge_{g \in \mathcal{G}}\ \Omega_g)$	§7
$\surd\surd$	Global	**if** $\mathcal{F} = \varnothing$ **then** $\mu \wedge (\wedge_{g,h \in \mathcal{G}}\ \Omega_{g \cap h})$ **else** $\geq \mu \wedge (\wedge_{g,h \in \mathcal{G}}\ \Omega_{g \cap h})$	§6.2

This result states that genuine atomic multicast requires some form of perfect failure detection in intersecting groups. Consequently, almost all protocols published to date (e.g., [30, 17, 20, 10, 29, 13]) assume the existence of such a decomposition.

Motivation. A key observation is that the impossibility result in [25] is established when atomic multicast allows a message to be disseminated to *any* subset of the processes. However, where there is no such need, weaker synchrony assumptions may just work. For instance, when each message is addressed to a single process, the problem is trivial and can be solved in an asynchronous system. Conversely, when every message is addressed to all the processes in the system, atomic multicast boils down to atomic broadcast, and thus ultimately to consensus. Now, if no two groups intersect, solving consensus inside each group seems both necessary and sufficient. In this paper, we further push this line of thought to characterize the necessary and sufficient synchrony assumptions to solve genuine atomic multicast.

Our results are established in the unreliable failure detectors model [9, 18]. A failure detector is an oracle available locally to each process that provides information regarding the speed at which the other processes are taking steps. Finding the weakest failure detector to solve a given problem is a central question in distributed computing literature [18]. In particular, the seminal work in [8] shows that a leader oracle (Ω) is the weakest failure detector for consensus when a majority of processes is correct. If any processes might fail, then a quorum failure detector (Σ) is required in addition to Ω [15].

A failure detector is realistic when it cannot guess the future. In [14], the authors prove that the perfect failure detector (P) is the weakest realistic failure detector to solve consensus. Building upon this result, Schiper and Pedone [34] shows that P is sufficient to implement genuine atomic multicast. However, P is the weakest only when messages are addressed to all the processes in the system. The present paper generalizes this result and the characterization given in [25] (see Table 1). It establishes the weakest failure detector to solve genuine atomic multicast for any set of destination groups.

Primer on the findings. Let $\mathcal{G}$ be all the destination groups and $\mathcal{F}$ be the cyclic families in it, that is the subsets of $\mathcal{G}$ whose intersection graph is hamiltonian. This paper shows that the weakest failure detector to solve genuine atomic multicast is $\mu = (\wedge_{g,h \in \mathcal{G}}\ \Sigma_{g \cap h}) \wedge (\wedge_{g \in \mathcal{G}}\ \Omega_g) \wedge \gamma$, where Σ_P and Ω_P are the quorum and leader failure detectors restricted to the processes in P, and γ is a new failure detector that informs the processes in a cyclic family $\mathfrak{f} \in \mathcal{F}$ when $\mathfrak{f}$ is faulty. Our results regarding γ are established wrt. realistic failure detectors.

This paper also studies two classical variations of the atomic multicast problem. The strict variation requires that message delivery follows the real-time order. In this case, we prove that μ must be strengthen with $1^{g \cap h}$, the indicator failure detector that informs each

process in $g \cup h$ when $g \cap h$ is faulty. The strongly genuine variation requires a message to be delivered when its destination group runs in isolation. In that case, the weakest failure detector is at least $\mu \wedge (\wedge_{g,h \in \mathcal{G}} \Omega_{g \cap h})$. This value is attained when $\mathcal{F} = \varnothing$.

Outline of the paper. §2 introduces the atomic multicast problem and the notion of genuineness. We present the candidate failure detector in §3. §4 proves that this candidate is sufficient. Its necessity is established in §5. §6 details the results regarding the two variations of the problem. We cover related work and discuss our results in §7. §8 closes this paper. Due to space constraints, all the proofs are deferred to the extended version [37].

2 The Atomic Multicast Problem

2.1 System Model

In [9], the authors extend the usual model of asynchronous distributed computation to include failure detectors. The present paper follows this model with the simplifications introduced in [23, 22]. This model is recalled in [37].

2.2 Problem Definition

Atomic multicast is a group communication primitive that allows to disseminate messages between processes. This primitive is used to build transactional systems [11, 36] and partially-replicated (aka., sharded) data stores [17, 32]. In what follows, we consider the most standard definition for this problem [4, 26, 12]. In the parlance of Hadzilacos and Toueg [26], it is named uniform global total order multicast. Other variations are studied in §6.

Given a set of messages $\mathcal{M}$, the interface of atomic multicast consists of operations $\mathit{multicast}(m)$ and $\mathit{deliver}(m)$, with $m \in \mathcal{M}$. Operation $\mathit{multicast}(m)$ allows a process to *multicast* a message m to a set of processes denoted by $\mathit{dst}(m)$. This set is named the *destination group* of m. When a process executes $\mathit{deliver}(m)$, it delivers message m, typically to an upper applicative layer.

Consider two messages m and m' and some process $p \in \mathit{dst}(m) \cap \mathit{dst}(m')$. Relation $m \overset{p}{\mapsto} m'$ captures the local delivery order at process p. This relation holds when, at the time p delivers m, p has not delivered m'. The union of the local delivery orders gives the *delivery order*, that is $\mapsto = \cup_{p \in \mathcal{P}} \overset{p}{\mapsto}$. The runs of atomic multicast must satisfy:

(Integrity) For every process p and message m, p delivers m at most once, and only if p belongs to $\mathit{dst}(m)$ and m was previously multicast.

(Termination) For every message m, if a correct process multicasts m, or a process delivers m, eventually every correct process in $\mathit{dst}(m)$ delivers m.

(Ordering) The transitive closure of $\mapsto$ is a strict partial order over $\mathcal{M}$.

Integrity and termination are two common properties in group communication literature. They respectively ensure that only sound messages are delivered to the upper layer and that the communication primitive makes progress. Ordering guarantees that the messages could have been received by a sequential process. A common and equivalent rewriting of this property is as follows:

(Ordering) Relation $\mapsto$ is acyclic over $\mathcal{M}$.

If the sole destination group is $\mathcal{P}$, that is the set of all the processes, the definition above is the one of atomic broadcast. In what follows, $\mathcal{G} \subseteq 2^{\mathcal{P}}$ is the set of all the destinations groups, i.e., $\mathcal{G} = \{g : \exists m \in \mathcal{M}.\, g = dst(m)\}$. For some process p, $\mathcal{G}(p)$ denotes the destination groups in $\mathcal{G}$ that contain p. Two groups g and h are *intersecting* when $g \cap h \neq \varnothing$.

What can be sent and to who. The process that executes $multicast(m)$ is the sender of m, denoted $src(m)$. As usual, we consider that processes disseminate different messages (i.e., src is a function). A message holds a bounded payload $payload(m)$, and we assume that atomic multicast is not payload-sensitive. This means that for every message m, and for every possible payload x, there exists a message $m' \in \mathcal{M}$ such that $payload(m') = x$, $dst(m') = dst(m)$ and $src(m') = src(m)$.

Dissemination model. In this paper, we consider a closed model of dissemination. This means that to send a message to some group g, a process must belong to it (i.e., $src(m) \in dst(m)$). In addition, we do not restrict the source of a message. This translates into the fact that for every message m, for every process p in $dst(m)$, there exists a message m' with $dst(m) = dst(m')$ and $src(m') = p$. Under the above set of assumptions, the atomic multicast problem is fully determined by the destination groups $\mathcal{G}$.

2.3 Genuineness

At first glance, atomic multicast boils down to the atomic broadcast problem: to disseminate a message it suffices to broadcast it, and upon reception only messages addressed to the local machine are delivered. With this approach, every process takes computational steps to deliver every message, including the ones it is not concerned with. As a consequence, the protocol does not scale [31, 35], even if the workload is embarrassingly parallel (e.g., when the destinations groups are pairwise disjoint).

Such a strategy defeats the core purpose of atomic multicast and is thus not satisfying from a performance perspective. To rule out this class of solutions, Guerraoui and Schiper [25] introduce the notion of *genuine* atomic multicast. These protocols satisfy the minimality property defined below.

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

All the results stated in this paper concern genuine atomic multicast. To date, this is the most studied variation for this problem (see, e.g., [30, 20, 10]).

3 The Candidate Failure Detector

This paper characterizes the weakest failure detector to solve genuine atomic multicast. Below, we introduce several notions related to failure detectors then present our candidate.

Family of destination groups. A family of destination groups is a set of (non-repeated) destination groups $\mathfrak{f} = (g_i)_i$. For some family $\mathfrak{f}$, $cpaths(\mathfrak{f})$ are the closed paths in the intersection graph of $\mathfrak{f}$ visiting all its destination groups.[1] Family $\mathfrak{f}$ is *cyclic* when its intersection graph is hamiltonian, that is when $cpaths(\mathfrak{f})$ is non-empty. A cyclic family $\mathfrak{f}$ is *faulty at time t* when every path $\pi \in cpaths(\mathfrak{f})$ visits an edge (g, h) with $g \cap h$ faulty at t.

[1] The intersection graph of a family of sets $(S_i)_i$ is the undirected graph whose vertices are the sets S_i, and such that there is an edge linking S_i and S_j iff $S_i \cap S_j \neq \varnothing$.

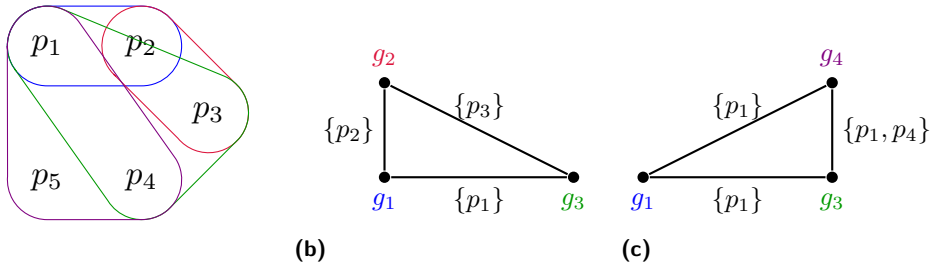

Figure 1 From left to right: the four groups g_1, g_2, g_3 and g_4, and the intersection graphs of the two cyclic families $\mathfrak{f} = \{g_1, g_2, g_3\}$ and $\mathfrak{f}' = \{g_1, g_3, g_4\}$.

In what follows, $\mathcal{F}$ denotes all the cyclic families in $2^{\mathcal{G}}$. Given a destination group g, $\mathcal{F}(g)$ are the cyclic families in $\mathcal{F}$ that contain g. For some process p, $\mathcal{F}(p)$ are the cyclic families $\mathfrak{f}$ such that p belongs to some group intersection in $\mathfrak{f}$ (that is, $\exists g, h \in \mathfrak{f}.\, p \in g \cap h$).

To illustrate the above notions, consider Figure 1. In this figure, $\mathcal{P} = \{p_1, \ldots, p_5\}$ and we have four destination groups: $g_1 = \{p_1, p_2\}$, $g_2 = \{p_2, p_3\}$, $g_3 = \{p_1, p_3, p_4\}$ and $g_4 = \{p_1, p_4, p_5\}$. The intersection graphs of $\mathfrak{f} = \{g_1, g_2, g_3\}$ and $\mathfrak{f}' = \{g_1, g_3, g_4\}$ are depicted respectively in Figures 1b and 1c. These two families are cyclic. This is also the case of $\mathfrak{f}'' = \mathcal{G} = \{g_1, g_2, g_3, g_4\}$ whose intersection graph is the union of the two intersection graphs of $\mathfrak{f}$ and $\mathfrak{f}'$. This family is faulty when $g_2 \cap g_1 = \{p_2\}$ fails. Group g_2 belongs to two cyclic families, namely $\mathcal{F}(g_2) = \{\mathfrak{f}, \mathfrak{f}''\}$. Process p_1 belongs to all cyclic families, that is $\mathcal{F}(p_1) = \mathcal{F}$. Differently, since p_5 is part of no group intersection, $\mathcal{F}(p_5) = \varnothing$.

Failure detectors of interest. Failure detectors are grouped into classes of equivalence that share common computational power. Several classes of failure detectors have been proposed in the past. This paper makes use of two common classes of failure detectors, Σ and Ω, respectively introduced in [15] and [8]. We also propose a new class γ named the *cyclicity failure detector*. All these classes are detailed below.

- The quorum failure detector (Σ) captures the minimal amount of synchrony to implement an atomic register. When a process p queries at time t a detector of this class, it returns a non-empty subset of processes $\Sigma(p,t) \subseteq \mathcal{P}$ such that:
  (Intersection) $\forall p, q \in \mathcal{P}.\, \forall t, t' \in \mathbb{N}.\, \Sigma(p,t) \cap \Sigma(q,t') \neq \varnothing$
  (Liveness) $\forall p \in \mathit{Correct}.\, \exists \tau \in \mathbb{N}.\, \forall t \geq \tau.\, \Sigma(p,t) \subseteq \mathit{Correct}$
  The first property states that the values of any two quorums taken at any times intersect. It is used to maintain the consistency of the atomic register. The second property ensures that eventually only correct processes are returned.
- Failure detector Ω returns an eventually reliable leader [16]. In detail, it returns a value $\Omega(p,t) \in \mathcal{P}$ satisfying that:
  (Leadership) $\mathit{Correct} \neq \varnothing \Rightarrow (\exists l \in \mathit{Correct}.\, \forall p \in \mathit{Correct}.\, \exists \tau \in \mathbb{N}.\, \forall t \geq \tau.\, \Omega(p,t) = l)$
  Ω is the weakest failure detector to solve consensus when processes have access to a shared memory. For message-passing distributed systems, $\Omega \wedge \Sigma$ is the weakest failure detector.
- The cyclicity failure detector (γ) informs each process of the cyclic families it is currently involved with. In detail, failure detector γ returns at each process p a set of cyclic families $\mathfrak{f} \in \mathcal{F}(p)$ such that:
  (Accuracy) $\forall p \in \mathcal{P}.\, \forall t \in \mathbb{N}.\, (\mathfrak{f} \in \mathcal{F}(p) \wedge \mathfrak{f} \notin \gamma(p,t)) \Rightarrow \mathfrak{f}$ faulty at t
  (Completeness) $\forall p \in \mathit{Correct}.\, \forall t \in \mathbb{N}.\, (\mathfrak{f} \in \mathcal{F}(p) \wedge \mathfrak{f}$ faulty at $t) \Rightarrow \exists \tau \in \mathbb{N}.\, \forall t' \geq \tau.\, \mathfrak{f} \notin \gamma(p,t')$

Accuracy ensures that if some cyclic family $\mathfrak{f}$ is not output at p and p belongs to it, then $\mathfrak{f}$ is faulty at that time. Completeness requires that eventually γ does not output forever a faulty family at the correct processes that are part of it. Hereafter, $\gamma(g)$ denotes the groups h such that $g \cap h \neq \varnothing$ and g and h belong to a cyclic family output by γ.

To illustrate the above definitions, we may consider again the system depicted in Figure 1. Let us assume that $\mathit{Correct} = \{p_1, p_4, p_5\}$. The quorum failure detector Σ can return g_1 or g_3, then g_4 forever. Failure detector Ω may output any process, then at some point in time, one of the correct processes (e.g., p_1) ought to be elected forever. At processes p_1, γ returns initially $\{\mathfrak{f}, \mathfrak{f}', \mathfrak{f}''\}$. Then, once families $\mathfrak{f}$ and $\mathfrak{f}''$ are faulty – this should happen as p_2 is faulty – the output eventually stabilizes to $\{\mathfrak{f}'\}$. When this happens, $\gamma(g_1) = \{g_3, g_4\}$.

Conjunction of failure detectors. We write $C \wedge D$ the conjunction of the failure detectors C and D [23]. For a failure pattern F, failure detector $C \wedge D$ returns a history in $D(F) \times C(F)$.

Set-restricted failure detectors. For some failure detector D, D_P is the failure detector obtained by restricting D to the processes in $P \subseteq \mathcal{P}$. This failure detector behaves as D for the processes $p \in P$, and it returns $\perp$ at $p \notin P$. In detail, let $F \cap P$ be the failure pattern F obtained from F by removing the processes outside P, i.e., $(F \cap P)(t) = F(t) \cap P$. Then, $D_P(F)$ equals $D(F \cap P)$ at $p \in P$, and the mapping $p \times \mathbb{N} \to \perp$ elsewhere. To illustrate this definition, $\Omega_{\{p\}}$ is the trivial failure detector that returns p at process p. Another example is given by $\Sigma_{\{p_1,p_2\}}$ which behaves as Σ over $\mathcal{P} = \{p_1, p_2\}$.

The candidate. Our candidate failure detector is $\mu_{\mathcal{G}} = (\wedge_{g,h \in \mathcal{G}}\ \Sigma_{g \cap h}) \wedge (\wedge_{g \in \mathcal{G}}\ \Omega_g) \wedge \gamma$. When the set of destinations groups $\mathcal{G}$ is clear from the context, we shall omit the subscript.

4 Sufficiency

This section shows that genuine atomic multicast is solvable with the candidate failure detector. A first observation toward this result is that consensus is wait-free solvable in g using $\Sigma_g \wedge \Omega_g$. Indeed, Σ_g permits to build shared atomic registers in g [15]. From these registers, we may construct an obstruction-free consensus and boost it with Ω_g [24]. Thus, any linearizable wait-free shared objects is implementable in g [27]. Leveraging these observations, this section depicts a solution built atop (high-level) shared objects.

Below, we first introduce group sequential atomic multicast (§4.1). From a computability perspective, this simpler variation is equivalent to the common atomic multicast problem. This is the variation that we shall implement hereafter. We then explain at coarse grain how to solve genuine atomic multicast in a fault-tolerant manner using the failure detector μ (§4.2). Further, the details of our solution are presented and its correctness informally argued (§4.3).

4.1 A simpler variation

Group sequential atomic multicast requires that each group handles its messages sequentially. In detail, given two messages m and m' addressed to the same group, we write $m \prec m'$ when $\mathit{src}(m')$ delivers m before it multicasts m'. This variation requires that if m and m' are multicast to the same group, then $m \prec m'$, or the converse, holds. Proposition 1 below establishes that this variation is as difficult as (vanilla) atomic multicast. Building upon this insight, this section depicts a solution to group sequential atomic multicast using failure detector μ.

▶ **Proposition 1.** *Group sequential atomic multicast is equivalent to atomic multicast.*

4.2 Overview of the solution

First of all, we observe that if the groups are pairwise disjoint, it suffices that each group orders the messages it received to solve atomic multicast. To this end, we use a shared log LOG_g per group g. Then, consider that the intersection graph of $\mathcal{G}$ is acyclic, i.e., $\mathcal{F}$ is empty, yet groups may intersect. In that case, it suffices to add a deterministic merge procedure in each group intersection, for instance, using a set of logs $\text{LOG}_{g \cap h}$ when $g \cap h \neq \varnothing$.

Now, to solve the general case, cycles in the order built with the shared logs must be taken into account. To this end, we use a fault-tolerant variation of Skeen's solution [5, 21]: in each log, the message is bumped to the highest initial position it occupies in all the logs. In the original algorithm [5], as in many other approaches (e.g., [20, 10]), such a procedure is failure-free, and processes simply agree on the final position (aka., timestamp) of the message in the logs. In contrast, our algorithm allows a disagreement when the cyclic family becomes faulty. This disagreement is however restricted to different logs, as in the acyclic case.

4.3 Algorithm

Algorithm 1 depicts a solution to (group sequential) genuine atomic multicast using failure detector μ. To the best of our knowledge, this is only algorithm with [34] that tolerates arbitrary failures. Algorithm 1 is composed of a set of actions. An action is executable once its preconditions (**pre:**) are true. The effects (**eff:**) of an action are applied sequentially until it returns. Algorithm 1 uses a log per group and per group intersection. Logs are linearizable, long-lived and wait-free. Their sequential interface is detailed below.

Logs. A *log* is an infinite array of slots. Slots are numbered from 1. Each slot contains one or more data items. A datum d is at position k when slot k contains it. This position is obtained through a call to $pos(d)$; 0 is returned if d is absent. A slot k is *free* when it contains no data item. In the initial state, every slot is free. The head of the log points to the first free slot after which there are only free slots (initially, slot 1). Operation $append(d)$ inserts datum d at the slot pointed by the head of the log then returns its position. If d is already in the log, this operation does nothing. When d is in the log, it can be locked with operation $bumpAndLock(d, k)$. This operation moves d from its current slot l to slot $\max(k, l)$, then locks it. Once locked, a datum cannot be bumped anymore. Operation $locked(d)$ indicates if d is locked in the log. We write $d \in L$ when datum d is at some position in the log L. A log implies an ordering on the data items it contains. When d and d' both appear in L, $d <_L d'$ is true when the position of d is lower than the position of d', or they both occupy the same slot and $d < d'$, for some a priori total order $(<)$ over the data items.

Variables. Algorithm 1 employs two types of shared objects at a process. First, for any two groups h and h' to which the local process belongs, Algorithm 1 uses a log $\text{LOG}_{h \cap h'}$ (line 2). Notice that, when $h = h'$, the log coincides with the log of the destination group h, i.e., LOG_h. Second, to agree on the final position of a message, Algorithm 1 also employs consensus objects (line 3). Consensus objects are both indexed by messages and group families. Given some message m and appropriate family $\mathfrak{f}$, Algorithm 1 calls $\text{CONS}_{m,\mathfrak{f}}$ (lines 20 and 21). Two processes call the same consensus object at line 21 only if both parameters match. Finally, to store the status of messages addressed to the local process, Algorithm 1 also employs a mapping PHASE (line 4). A message is initially in the `start` phase, then it moves to `pending` (line 15), `commit` (line 24), `stable` (line 33) and finally the `deliver` (line 37) phase. Phases are ordered according to this progression.

Algorithm 1 Solving atomic multicast with failure detector μ – code at process p.

```
1: variables:
2:     (LOG_{h∩h'})_{h,h'∈G(p)}
3:     (CONS_{m,f})_{m∈M,f⊆G}
4:     PHASE[m] ← λm.start

5: multicast(m) :=                                   // g = dst(m) ∧ g ∈ G(p)
6:     pre: PHASE[m] = start
7:     eff: LOG_g.append(m)

8: pending(m) :=
9:     pre: PHASE[m] = start
10:         m ∈ LOG_g
11:         ∀m' <_{LOG_g} m. PHASE[m'] ≥ commit
12:    eff: for all h ∈ G(p) do
13:             i ← LOG_{g∩h}.append(m)
14:             LOG_g.append(m, h, i)
15:         PHASE[m] ← pending

16: commit(m) :=
17:    pre: PHASE[m] = pending
18:         ∀h ∈ γ(g). (m, h, −) ∈ LOG_g
19:    eff: let k = max{i : ∃(m, −, i) ∈ LOG_g},
20:         let f = {h : ∃f' ∈ F(p). g, h ∈ f' ∧ g ∩ h ≠ ∅}
21:         k ← CONS_{m,f}.propose(k)
22:         for all h ∈ G(p) do
23:             LOG_{g∩h}.bumpAndLock(m, k)
24:         PHASE[m] ← commit

25: stabilize(m, h) :=
26:    pre: PHASE[m] = commit
27:         h ∈ G(p)
28:         ∀m' <_{LOG_{g∩h}} m. PHASE[m'] ≥ stable
29:    eff: LOG_g.append(m, h)

30: stable(m) :=
31:    pre: PHASE[m] = commit
32:         ∀h ∈ γ(g). (m, h) ∈ LOG_g
33:    eff: PHASE[m] ← stable

34: deliver(m) :=
35:    pre: PHASE[m] = stable
36:         ∀m' <_{LOG_{g∩h}} m. PHASE[m'] = deliver
37:    eff: PHASE[m] ← deliver
```

Algorithmic details. We now detail Algorithm 1 and jointly argue about its correctness. For clarity, our argumentation is informal – the full proof appears in [37].

To multicast some message m to $g = dst(m)$, a process adds m to the log of its destination group (line 7). When $p \in g$ observes m in the log, p appends m to each $\mathrm{LOG}_{g\cap h}$ with $p \in g \cap h$ (line 13). Then, p stores in the log of the destination group of m the slot occupied by m in $\mathrm{LOG}_{g\cap h}$ (line 14). This moves m to the `pending` phase.

Similarly to Skeen's algorithm [5], a message is then bumped to the highest slot it occupies in the logs. This step is executed at lines 16-24. In detail, p first agrees with its peers on the highest position k occupied by m (lines 19-21). Observe here that only the processes

in g that share some cyclic family with p take part to this agreement (line 20). Then, for each group h in $\mathcal{G}(p)$, p bumps m to slot k in $\text{LOG}_{g\cap h}$ and locks it in this position (line 23). This moves m to the `commit` phase.

The next steps of Algorithm 1 compute the predecessors of message m. With more details, once m reaches the `stable` phase and is ready to be delivered, the messages that precede it in the logs at process p cannot change anymore.

If g does not belong to any cyclic family, stabilizing m is immediate: the precondition at line 32 is always vacuously true. In this case, m is delivered in an order consistent with the order it is added to the logs (line 28). This comes from the fact that when $\mathcal{F} = \varnothing$ ordering the messages reduces to a pairwise agreement between the processes.

When $\mathcal{F} \neq \varnothing$, stabilizing m is a bit more involved. Indeed, messages can be initially in cyclic positions, e.g., $C = m_1 <_{\text{LOG}_{g_1\cap g_2}} m_2 <_{\text{LOG}_{g_2\cap g_3}} m_3 <_{\text{LOG}_{g_3\cap g_1}} m_1$, preventing them to be delivered. As in [5], bumping messages helps to resolve such a situation.

The bumping procedure is executed globally. A process must wait that the positions in the logs of a message are cycle-free before declaring it `stable`. Waiting can cease when the cyclic family is faulty (line 32). This is correct because messages are stabilized in the order of their positions in the logs (lines 25-29). Hence, if a cycle C exists initially in the positions, either (i) not all the messages in C are delivered, or (ii) the first message to get `stable` in C has no predecessors in C in the logs. In other words, for any two messages m and m' in C, if $m \mapsto m'$ then m is `stable` before m'.

A process indicates that message m with $g = dst(m)$ is stabilized in group h with a pair (m, h) in LOG_g (line 29). When this holds for all the groups h intersecting with g such that there exists a correct family $\mathfrak{f}$ with $f \in \mathcal{F}(p)$ and $g, h \in \mathfrak{f}$, m is declared `stable` at p (line 32). Once `stable`, a message m can be delivered (lines 34-37).

Algorithm 1 stabilizes then delivers messages according to their positions in the logs. To maintain progress, these positions must remain acyclic at every correct process. Furthermore, this should also happen globally when a cyclic family is correct. Both properties are ensured by the calls to consensus objects (line 21).

Implementing the shared objects. In each group g, consensus is solvable since μ provides $\Sigma_g \wedge \Omega_g$. This serves to implement all the objects $(\text{CONS}_{m,\mathfrak{f}})_{m,\mathfrak{f}}$ when $dst(m) = g$. Logs that are specific to a group, namely $(\text{LOG}_g)_{g\in\mathcal{G}}$, are also built atop consensus in g using a universal construction [27].

Failure detector μ does not offer the means to solve consensus in $g \cap h$. Hence we must rely on either g or h to build $\text{LOG}_{g\cap h}$. Minimality requires processes in a destination group to take steps only in the case a message is addressed to them. To achieve this, we have to slightly modify the universal construction for $\text{LOG}_{g\cap h}$, as detailed next.

First, we consider that this construction relies on an unbounded list of consensus objects.[2] Each consensus object in this list is contention-free fast [2]. This means that it is guarded by an adopt-commit object (AC) [19] before an actual consensus object (CONS) is called. Upon calling *propose*, AC is first used and if it fails, that is "adopt" is returned, CONS is called. Adopt-commit objects are implemented using $\Sigma_{g\cap h}$, while consensus objects are implemented atop some group, say g, using $\Sigma_g \wedge \Omega_g$. This modification ensures that when processes execute operations in the exact same order, only the adopt-commit objects are called. As a consequence, when no message is addressed to either g or h during a run, only the processes in $g \cap h$ executes steps to implement an operation of $\text{LOG}_{g\cap h}$.

[2] In the failure detector model, computability results can use any amount of shared objects.

Algorithm 2 Emulating $\Sigma_{\cap_{g\in G} g}$ – code at process p.

1: **variables:**
2: $(A_{g,x})_{g\in G, x\subseteq g.p\in x}$
3: $(Q_g)_{g\in G} \leftarrow \lambda g.\{g\}$
4: $(qr_g)_{g\in G} \leftarrow \lambda g.g$
5: **for all** $g \in G, x \subseteq g : p \in x$ **do**
6: **let** m such that $dst(m) = g \wedge payload(m) = p$
7: $A_{g,x}.multicast(m)$
8: **when** $A_{g,x}.deliver(-)$
9: $Q_g \leftarrow Q_g \cup \{x\}$
10: **when** *query*
11: **if** $p \notin \bigcap_{g\in G} g$ **then**
12: **return** $\perp$
13: **for all** $g \in G$ **do**
14: $qr_g \leftarrow$ **choose** $\arg\max_{y\in Q_g} rank(y)$
15: **return** $(\bigcup_{g\in G} qr_g) \cap (\bigcap_{g\in G} g)$

5 Necessity

Consider some environment $\mathfrak{E}$, a failure detector D and an algorithm A that uses D to solve atomic multicast in $\mathfrak{E}$. This section shows that D is stronger than μ in $\mathfrak{E}$. To this end, we first use the fact that atomic multicast solves consensus per group. Hence μ is stronger than $\wedge_{g\in\mathcal{G}} (\Omega_g \wedge \Sigma_g)$. §5.1 proves that D is stronger than $\Sigma_{g\cap h}$ for any two groups $g, h \in \mathcal{G}$. Further, in §5.2, we establish that D is stronger than γ. This last result is established when D is realistic. The remaining cases are discussed in §7.

5.1 Emulating $\Sigma_{g\cap h}$

Atomic multicast solves consensus in each destination group. This permits to emulate $\wedge_{g\in\mathcal{G}}\Sigma_g$. However, for two intersecting groups g and h, $\Sigma_g \wedge \Sigma_h$ is not strong enough to emulate $\Sigma_{g\cap h}$.[3] Hence, we must build the failure detector directly from the communication primitive. Algorithm 2 presents such a construction. This algorithm can be seen as an extension of the work of Bonnet and Raynal [6] to extract Σ_k when k-set agreement is solvable. Algorithm 2 emulates $\Sigma_{\cap_{g\in G} g}$, where $G \subseteq \mathcal{G}$ is a set of at most two intersecting destination groups.

At a process p, Algorithm 2 employs multiple instances of algorithm A. In detail, for every group $g \in G$ and subset x of g, if process p belongs to x, then p executes an instance $A_{g,x}$ (line 2). Variable Q_g stores the responsive subsets of g, that is the sets $x \subseteq g$ for which $A_{g,x}$ delivers a message. Initially, this variable is set to $\{g\}$.

Algorithm 2 uses the ranking function defined in [6]. For some set $x \subseteq \mathcal{P}$, function $rank(x)$ outputs the rank of x. Initially, all the sets have rank 0. Function *rank* ensures a unique property: a set x is correct if and only if it ranks grows forever. To compute this function, processes keep track of each others by exchanging (asynchronously) "alive" messages. At a process p, the number of "alive" messages received so far from q defines the rank of q. The rank of a set is the lowest rank among all of its members.

[3] The two detectors may return forever non-intersecting quorums.

At the start of Algorithm 2, a process atomic multicasts its identity for every instance $A_{g,x}$ it is executing (line 7). When, $A_{g,x}$ delivers a process identity, x is added to variable Q_g (line 9). Thus, variable Q_g holds all the instances $A_{g,x}$ that progress successfully despite that $g \setminus x$ do not participate. From this set, Algorithm 2 computes the most responsive quorum using the ranking function (line 14). As stated in Theorem 2 below, these quorums must intersect at any two processes in $\cap_{g \in G} g$.

▶ **Theorem 2.** *Algorithm 2 implements* $\Sigma_{\cap_{g \in G} g}$ *in* $\mathfrak{E}$.

5.2 Emulating γ

Target systems. A process p is failure-prone in environment $\mathfrak{E}$ when for some failure pattern $F \in \mathfrak{E}$, $p \in \mathit{Faulty}(F)$. By extension, we say that $P \subseteq \mathcal{P}$ is failure-prone when for some $F \in \mathfrak{E}$, $P \subseteq \mathit{Faulty}(F)$. A cyclic family $\mathfrak{f}$ is failure-prone when one of its group intersections is failure-prone. Below, we consider that $\mathfrak{E}$ satisfies that if a process may fail, it may fail at any time (formally, $\forall F \in \mathfrak{E}. \forall p \in \mathit{Faulty}(F). \exists F' \in \mathfrak{E}. \forall t \in \mathbb{N}. \forall t' < t. F'(t') = F(t') \wedge F'(t) = F(t) \cup \{p\}$). We also restrict our attention to realistic failure detectors, that is they cannot guess the future [14].

Additional notions. Consider a cyclic family $\mathfrak{f}$. Two closed paths π and π' in $\mathit{cpaths}(\mathfrak{f})$ are equivalent, written $\pi \equiv \pi'$, when they visit the same edges in the intersection graph. A closed path π in $\mathit{cpaths}(\mathfrak{f})$ is oriented. The direction of π is given by $\mathit{dir}(\pi)$. It equals 1 when the path is clockwise, and -1 otherwise (for some canonical representation of the intersection graph). To illustrate these notions, consider family $\mathfrak{f}$ in Figure 1b. The sequence $\pi = g_3 g_1 g_2 g_3$ is a closed path in its intersection graph, with $|\pi| = 4$ and $\pi[0] = \pi[|\pi| - 1] = g_3$. The direction of this path is 1 since it is visiting clockwise the intersection graph of $\mathfrak{f}$ in the figure. Path π is equivalent to the path $\pi' = g_1 g_3 g_2 g_1$ which visits $\mathfrak{f}$ in the converse direction.

Construction. We emulate failure detector γ in Algorithm 3. For each closed path $\pi \in \mathit{cpaths}(\mathfrak{f})$ with $\pi[0] \cap \pi[1]$ failure-prone in $\mathfrak{E}$, Algorithm 3 maintains two variables: an instance A_π of the multicast algorithm A, and a flag $\mathit{failed}[\pi]$. Variable A_π is used to detect when a group intersection visited by π is faulty. It this happens, the flag $\mathit{failed}[\pi]$ is raised. When for every path $\pi \in \mathit{cpaths}(\mathfrak{f})$, some path equivalent to π is faulty, Algorithm 3 ceases returning the family $\mathfrak{f}$ (line 16).

In Algorithm 3, for every path $\pi \in \mathit{cpaths}(\mathfrak{f})$, the processes in $\pi[0] \cap \pi[1]$ multicast their identities to $\pi[0]$ using instance A_π (lines 4 and 5). In this instance of A, all the processes in $\mathfrak{f}$ but the intersection $\pi[0] \cap \pi[|\pi| - 2]$ participate (line 2). As the path is closed, this corresponds to the intersection with the last group preceding the first group in the path.

When $p \in \pi[i] \cap \pi[i+1]$ delivers a message $(-, i)$, it signals this information to the other members of the family (line 9). Then, p multicasts its identity to $\pi[i+1]$ (line 10). This mechanism is repeated until the antepenultimate group in the path is reached (line 8). When such a situation occurs, the flag $\mathit{failed}[\pi]$ is raised (line 12). This might also happen earlier when a message is received for some path π' equivalent to π and visiting $\mathfrak{f}$ in the converse direction (line 13).

Below, we claim that Algorithm 3 is a correct emulation of failure detector γ.

▶ **Theorem 3.** *Algorithm 3 implements* γ *in* $\mathfrak{E}$.

Algorithm 3 Emulating γ – code at process p.

variables:
  $(A_\pi)_\pi$ // $\forall \mathfrak{f} \in \mathcal{F}(p). \forall \pi \in cpaths(\mathfrak{f}). p \notin \pi[0] \cap \pi[|\pi| - 2]$
  $failed[\pi] \leftarrow \lambda\pi.false$

for all $A_\pi : p \in \pi[0] \cap \pi[1]$ **do**
  $A_\pi.multicast(p, 0)$ to $\pi[0]$

$signal(\pi, i)$:=
  pre: $A_\pi.deliver(-, i)$
    $i < |\pi| - 2 \wedge p \in \pi[i + 1]$
  eff: $send(\pi, i)$ to $\mathfrak{f}$
    $A_\pi.multicast(p, i + 1)$ to $\pi[i + 1]$

$update(\pi)$:=
  pre: $\exists \pi' \equiv \pi. rcv(\pi, j) \wedge \vee\ j = |\pi| - 3$
    $\vee\ (rcv(\pi', 0) \wedge \pi[j] = \pi'[0] \wedge dir(\pi) = -dir(\pi'))$
  eff: $failed[\pi] \leftarrow true$

when *query*
  return $\{\mathfrak{f} \in \mathcal{F}(p) : \exists \pi \in cpaths(\mathfrak{f}). \forall \pi' \equiv \pi. failed[\pi'] = false\}$

6 Variations

This section explores two common variations of the atomic multicast problem. It shows that each variation has a weakest failure detector stronger than μ. The first variation requires messages to be ordered according to real time. This means that if m is delivered before m' is multicast, no process may deliver m' before m. In this case, we establish that the weakest failure detector must accurately detect the failure of a group intersection. The second variation demands each group to progress independently in the delivery of the messages. This property strengthens minimality because in a genuine solution a process may help others as soon as it has delivered a message. We show that the weakest failure detector for this variation permits to elect a leader in each group intersection.

6.1 Enforcing real-time order

Ordering primitives like atomic broadcast are widely used to construct dependable services [7]. The classical approach is to follow state-machine replication (SMR), a form of universal construction. In SMR, a service is defined by a deterministic state machine, and each replica maintains its own local copy of the machine. Commands accessing the service are funneled through the ordering primitive before being applied at each replica on the local copy.

SMR protocols must satisfy linearizability [28]. However, as observed in [3], the common definition of atomic multicast is not strong enough for this: if some command d is submitted after a command c get delivered, atomic multicast does not enforce c to be delivered before d, breaking linearizability. To sidestep this problem, a stricter variation must be used. Below, we define such a variation and characterize its weakest failure detector.

6.1.1 Definition

We write $m \rightsquigarrow m'$ when m is delivered in real-time before m' is multicast. Atomic multicast is *strict* when ordering is replaced with: (*Strict Ordering*) The transitive closure of $(\mapsto \cup \rightsquigarrow)$ is a strict partial order over $\mathcal{M}$. Strictness is free when there is a single destination group.

Algorithm 4 Emulating $1^{g\cap h}$ – code at process $p \in g \cup h$.

```
1: variables:
2:     B ← if (p ∈ g \ h) then A_g else if (p ∈ h \ g) then A_h else ⊥  // A_g and A_h are distinct instances of A
3:     failed ← false
4: if B ≠ ⊥ then
5:     B.multicast(p)
6:     wait until B.deliver(−)
7:     send(failed) to g ∪ h
8: when rcv(failed)
9:     failed ← true
10: when query
11:     return failed
```

Indeed, if p delivers m before q broadcasts m', then necessarily $m \overset{p}{\mapsto} m'$. This explains why atomic broadcast does not mention such a requirement. In what follows, we prove that strict atomic multicast is harder than (vanilla) atomic multicast.

6.1.2 Weakest failure detector

Candidate. For some (non-empty) group of processes P, the *indicator failure detector* 1^P indicates if all the processes in P are faulty or not. In detail, this failure detector returns a boolean which ensures that:

(Accuracy) $\forall p \in \mathcal{P}. \forall t \in \mathbb{N}. 1^P(p,t) \Rightarrow P \subseteq F(t)$
(Completeness) $\forall p \in \mathit{Correct}. \forall t \in \mathbb{N}. P \subseteq F(t) \Rightarrow \exists \tau \in \mathbb{N}. \forall t' \geq \tau. 1^P(p,t')$

For simplicity, we write $1^{g\cap h}$ the indicator failure detector restricted to the processes in $g \cup h$ (that is, the failure detector $1^{g\cap h}_{g\cup h}$). This failure detector informs the processes outside $g \cap h$ when the intersection is faulty. Notice that for the processes in the intersection, $1^{g\cap h}$ does not provide any useful information. This comes from the fact that simply returning always *true* is a valid implementation at these processes.

Our candidate failure detector is $\mu \wedge (\wedge_{g,h\in\mathcal{G}}\ 1^{g\cap h})$. One can establish that $\wedge_{g,h\in\mathcal{G}} 1^{g\cap h}$ is stronger than γ (see Proposition 4 below). As a consequence, this failure detector can be rewritten as $(\wedge_{g,h\in\mathcal{G}}\ \Sigma_{g\cap h} \wedge 1^{g\cap h}) \wedge (\wedge_{g\in\mathcal{G}}\ \Omega_g)$.

▶ **Proposition 4.** $\wedge_{g,h\in\mathcal{G}}\ 1^{g\cap h} \leq \gamma$

Necessity. An algorithm to construct $1^{g\cap h}$ is presented in Algorithm 4. It relies on an implementation A of strict atomic multicast that makes use internally of some failure detector D. Proposition 5 establishes the correctness of such a construction.

▶ **Proposition 5.** *Algorithm 4 implements* $1^{g\cap h}$.

Sufficiency. The solution to strict atomic multicast is almost identical to Algorithm 1. The only difference is at line 32 when a message moves to the `stable` phase. Here, for every destination group h with $h \cap g \neq \varnothing$, a process waits either that $1^{g\cap h}$ returns *true*, or that a tuple (m,h) appears in LOG_g. From Proposition 4, we know that the indicator failure detector $1^{g\cap h}$ provides a better information than γ regarding the correctness of $g \cap h$. As a consequence, the modified algorithm solves (group sequential) atomic multicast.

Now, to see why such a solution is strict, consider two messages m and m' that are delivered in a run, with $g = dst(m)$ and $h = dst(m')$. We observe that when $m' \rightsquigarrow m$ or $m' \mapsto m$, m' is stable before m, from which we deduce that strict ordering holds.

With more details, in the former case ($m' \rightsquigarrow m$), this comes from the fact that to be delivered a message must be `stable` first (line 35). In the later ($m' \mapsto m$), when message m is `stable` at some process p, p must wait a message (m, h) in LOG_g, or that $1^{g \cap h}$ returns *true*. If (m, h) is in LOG_g, then line 29 was called before by some process q. Because both messages are delivered and $m' \mapsto m$, m' must precedes m in $\text{LOG}_{g \cap h}$. Thus the precondition at line 28 enforces that m' is `stable` at q, as required. Now, if the indicator returns *true* at p, $m' \mapsto m$ tells us that a process delivers m' before m and this must happen before $g \cap h$ fails.

6.2 Improving parallelism

As motivated in the Introduction, genuine solutions to atomic multicast are appealing from a performance perspective. Indeed, if messages are addressed to disjoint destination groups in a run, they are processed in parallel by such groups. However, when contention occurs, a message may wait for a chain of messages to be delivered first. This chain can span outside of the destination group, creating a delay that harms performance and reduces parallelism [17, 1]. In this section, we explore a stronger form of genuineness, where groups are able to deliver messages independently. We prove that, similarly to the strict variation, this requirement demands more synchrony than μ from the underlying system.

6.2.1 Definition

As standard, a run R is fair for some correct process p when p executes an unbounded amount of steps in R. By extension, R is fair for $P \subseteq \mathit{Correct}(R)$, or for short P-fair, when it is fair for every p in P. If P is exactly the set of correct processes, we simply say that R is fair.

(Group Parallelism) Consider a message m and a run R. Note $P = \mathit{Correct}(R) \cap \mathit{dst}(m)$. If m is delivered by a process, or atomic multicast by a correct process in R, and R is P-fair, then every process in P delivers m in R.

Group parallelism bears similarity with x-obstruction freedom [38], in the sense that the system must progress when a small enough group of processes is isolated. A protocol is said *strongly genuine* when it satisfy both the minimality and the group parallelism properties.

6.2.2 About the weakest failure detector

Below, we establish that $(\wedge_{g,h \in \mathcal{G}}\ \Omega_{g \cap h})$ is necessary. It follows that the weakest failure detector for this variation is at least $\mu \wedge (\wedge_{g,h \in \mathcal{G}}\ \Omega_{g \cap h})$.

Emulating $\wedge_{g,h \in \mathcal{G}}\ \Omega_{g \cap h}$. Consider some algorithm A that solves strongly genuine atomic multicast with failure detector D. Using both A and D, each process may emulate $\Omega_{g \cap h}$, for some intersecting groups $g, h \in \mathcal{G}$. The emulation follows the general schema of CHT [8]. We sketch the key steps below. The full proof appears in [37].

Each process constructs a directed acyclic graph G by sampling the failure detector D and exchanging these samples with other processes. A path π in G induces multiple runs of A that each process locally simulates. A run starts from some initial configuration. In our context, the configurations $\mathfrak{I} = \{I_1, \ldots, I_{n \geq 2}\}$ of interest satisfy (i) the processes outside

$g \cap h$ do not atomic multicast any message, and (ii) the processes in $g \cap h$ multicast a single message to either g or h. For some configuration $I_i \in \mathfrak{I}$, the schedules of the simulated runs starting from I_i are stored in a simulation tree Υ_i. There exists an edge (S, S') when starting from configuration $S(I_i)$, one may apply a step $s = (p, m, d)$ for some process p, message m transiting in $S(I_i)$ and sample d of D such that $S' = S \cdot s$.

Every time new samples are received, the forest of the simulation trees $(\Upsilon_i)_i$ is updated. At each such iteration, the schedules in Υ_i are tagged using the following valency function: S is tagged with g (respectively, h) if for some successor S' of S in Υ_i a process in $g \cap h$ delivers first a message addressed to g (resp. to h) in configuration $S'(I_i)$. A tagged schedule is *univalent* when it has a single tag, and *bivalent* otherwise.

As the run progresses, each root of a simulation tree has eventually a stable set of tags. If the root of Υ_i is g-valent, the root of Υ_j is h-valent and they are adjacent, i.e., all the processes but some $p \in g \cap h$ are in the same state in I_i and I_j, then p must be correct. Otherwise, there exists a bivalent root of some tree Υ_i such that for g (respectively, h) a correct process multicasts a message to g (resp., h) in I_i. In this case, similarly to [8], there exists a decision gadget in the simulation tree Υ_i. This gadget is a sub-tree of the form (S, S', S''), with S bivalent, and S' g-valent and S'' h-valent (or vice-versa). Using the group parallelism property of A, we may then show that necessarily the deciding process in the gadget, that is the process taking a step toward either S' or S'' is correct and belongs to the intersection $g \cap h$.

Solution when $\mathcal{F} = \varnothing$. In this case, Algorithm 1 just works. To attain strong genuineness, each log object $\mathrm{LOG}_{g \cap h}$ is implemented with $\Sigma_{g \cap h} \wedge \Omega_{g \cap h}$ through standard universal construction mechanisms. When $\mathcal{F} = \varnothing$, $\mu \wedge (\wedge_{g,h \in \mathcal{G}}\ \Omega_{g \cap h})$ is thus the weakest failure detector. The case $\mathcal{F} \neq \varnothing$ is discussed in the next section.

7 Discussion

Several definitions for atomic multicast appear in literature (see, e.g., [12, 26] for a survey). Some papers consider a variation of atomic multicast in which the ordering property is replaced with: (*Pairwise Ordering*) If p delivers m then m', every process q that delivers m' has delivered m before. Under this definition, cycles in the delivery relation ($\mapsto$) across more than two groups are not taken into account. This is computably equivalent to $\mathcal{F} = \varnothing$. Hence the weakest failure detector for this variation is $(\wedge_{g,h \in \mathcal{G}}\ \Sigma_{g \cap h}) \wedge (\wedge_{g \in \mathcal{G}}\ \Omega_g)$.

In [25], the authors show that failure detectors of the class $\mathcal{U}_2$ are too weak to solve the pairwise ordering variation. These detectors can be wrong about (at least) two processes. In detail, the class $\mathcal{U}_k$ are all the failure detectors D that are k-unreliable, that is they cannot distinguish any pair of failure patterns F and F', as long as the faulty processes in F and F' are members of a subset W of size k (the "wrong" subset). The result in [25] is a corner case of the necessity of $\Sigma_{g \cap h}$ when $g \cap h = \{p, q\}$ and both processes are failure-prone in $\mathfrak{E}$. Indeed, $\Sigma_{\{p,q\}} \notin \mathcal{U}_2$. To see this, observe that if q is faulty and p correct, then $\{p\}$ is eventually the output of $\Sigma_{\{p,q\}}$ at p. A symmetrical argument holds for process q in runs where q is correct and p faulty. In the class $\mathcal{U}_2$, such values can be output in runs where both processes are correct, contradicting the intersection property of $\Sigma_{\{p,q\}}$.

Most atomic multicast protocols [30, 17, 20, 10, 31, 29, 13, 33] sidestep the impossibility result in [25] by considering that destination groups are decomposable into a set of disjoint groups, each of these behaving as a logically correct entity. This means that there exists a partitioning $\mathfrak{P}(\mathcal{G}) \subseteq 2^{\mathcal{P}}$ satisfying that (i) for every destination group $g \in \mathcal{G}$, there exists

$(g_i)_i \subseteq \mathfrak{P}(\mathcal{G})$ with $g = \cup_i g_i$, (ii) each $g \in \mathfrak{P}(\mathcal{G})$ is correct, and (iii) for any two g, h in $\mathfrak{P}(\mathcal{G})$, $g \cap h$ is empty. Since $\wedge_{g \in \mathfrak{P}(\mathcal{G})} (\Sigma_g \wedge \Omega_g) \succeq \mu$, we observe that solving the problem over $\mathfrak{P}(\mathcal{G})$ is always as difficult as over $\mathcal{G}$. It can also be more demanding in certain cases, e.g., if two groups intersect on a single process p, then p must be reliable. In Figure 1, this happens with process p_2. In contrast, to these prior solutions, Algorithm 1 tolerates any number of failures. This is also the case of [34] which relies on a perfect failure detector.

Regarding strongly genuine atomic multicast, §6.2 establishes that $\mu \wedge (\wedge_{g,h \in \mathcal{G}} \Omega_{g \cap h})$ is the weakest when $\mathcal{F} = \varnothing$. The case $\mathcal{F} \neq \varnothing$ is a bit more intricate. First of all, we may observe that in this case the problem is failure-free solvable: given a spanning tree T of the intersection graph of $\mathcal{G}$, we can deliver the messages according to the order $<_T$, that is, if m is addressed to g intersecting with $h, h', \ldots$ with $h <_T h' <_T \ldots$, then $g \cap h$ delivers first m, followed by $g \cap h'$, etc.[4] A failure-prone solution would apply the same logic. This is achievable using $\mu \wedge (\wedge_{g,h \in \mathcal{G}} \Omega_{g \cap h}) \wedge (\wedge_{g,h \in \mathcal{F}} 1^{g \cap h})$, where $g \in \mathcal{F}$ holds when for some family $\mathfrak{f} \in \mathcal{F}$, we have $g \in \mathfrak{f}$. We conjecture that this failure detector is actually the weakest.

8 Conclusion

This paper presents the first solution to genuine atomic multicast that tolerates arbitrary failures without using system-wide perfect failure detection. It also introduces two new classes of failure detectors: (γ) which tracks when a cyclic family of destination groups is faulty, and $(1^{g \cap h})$ that indicates when the group intersection $g \cap h$ is faulty. Building upon these new abstractions, we identify the weakest failure detector for genuine atomic multicast and also for several key variations of this problem. Our results offer a fresh perspective on the solvability of genuine atomic multicast in crash-prone systems. In particular, they question the common assumption of partitioning the destination groups. This opens an interesting avenue for future research on the design of fault-tolerant atomic multicast protocols.

References

1 Tarek Ahmed-Nacer, Pierre Sutra, and Denis Conan. The convoy effect in atomic multicast. In *35th IEEE Symposium on Reliable Distributed Systems Workshops, SRDS 2016 Workshop, Budapest, Hungary, September 26, 2016*, pages 67–72. IEEE Computer Society, 2016. doi:10.1109/SRDSW.2016.22.

2 Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov. Computing with reads and writes in the absence of step contention. In Pierre Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, volume 3724 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2005. doi:10.1007/11561927_11.

3 Carlos Eduardo Benevides Bezerra, Fernando Pedone, and Robbert van Renesse. Scalable state-machine replication. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 331–342. IEEE Computer Society, 2014. doi:10.1109/DSN.2014.41.

4 Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, August 1991. doi:10.1145/128738.128742.

5 Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computers Systems*, 5(1):47–76, January 1987. doi:10.1145/7351.7478.

[4] Strictly speaking, a spanning tree is required per connected component of the intersection graph.

6 François Bonnet and Michel Raynal. Looking for the weakest failure detector for k-set agreement in message-passing systems: Is π_k the end of the road? In *Stabilization, Safety, and Security of Distributed Systems, 11th International Symposium, SSS 2009, Lyon, France, November 3-6, 2009. Proceedings*, pages 149–164, 2009. doi:10.1007/978-3-642-05118-0_11.

7 Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In Indranil Gupta and Roger Wattenhofer, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 398–407. ACM, 2007. doi:10.1145/1281100.1281103.

8 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996. doi:10.1145/234533.234549.

9 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996. doi:10.1145/226643.226647.

10 Paulo R. Coelho, Nicolas Schiper, and Fernando Pedone. Fast atomic multicast. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pages 37–48. IEEE Computer Society, 2017. doi:10.1109/DSN.2017.15.

11 James A. Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 223–235. USENIX Association, 2012. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/cowling.

12 Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004. doi:10.1145/1041680.1041682.

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

14 Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. A realistic look at failure detectors. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*, pages 345–353. IEEE Computer Society, 2002. doi:10.1109/DSN.2002.1028919.

15 Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 338–346, 2004. doi:10.1145/1011767.1011818.

16 Swan Dubois, Rachid Guerraoui, Petr Kuznetsov, Franck Petit, and Pierre Sens. The weakest failure detector for eventual consistency. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 375–384, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2767386.2767404.

17 Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. Efficient replication via timestamp stability. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 178–193. ACM, 2021. doi:10.1145/3447786.3456236.

18 Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2), February 2011. doi:10.1145/1883612.1883616.

19 Eli Gafni. Round-by-round fault detectors (extended abstract): Unifying synchrony and asynchrony. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 143–152, New York, NY, USA, 1998. ACM. doi:10.1145/277697.277724.

20 Alexey Gotsman, Anatole Lefort, and Gregory V. Chockler. White-box atomic multicast. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 176–187. IEEE, 2019. doi:10.1109/DSN.2019.00030.

21 R. Guerraoui and A. Schiper. Total order multicast to multiple groups. In *Proceedings of 17th International Conference on Distributed Computing Systems*, pages 578–585, 1997. doi:10.1109/ICDCS.1997.603426.

22 Rachid Guerraoui, Vassos Hadzilacos, Petr Kuznetsov, and Sam Toueg. The weakest failure detectors to solve quittable consensus and nonblocking atomic commit. *SIAM J. Comput.*, 41(6):1343–1379, 2012. doi:10.1137/070698877.

23 Rachid Guerraoui, Maurice Herlihy, Petr Kouznetsov, Nancy Lynch, and Calvin Newport. On the weakest failure detector ever. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 235–243, New York, NY, USA, 2007. ACM. doi:10.1145/1281100.1281135.

24 Rachid Guerraoui and Michel Raynal. The alpha of indulgent consensus. *Comput. J.*, 50(1):53–67, 2007. doi:10.1093/comjnl/bxl046.

25 Rachid Guerraoui and André Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, 254(1-2):297–316, 2001. doi:10.1016/S0304-3975(99)00161-9.

26 Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

27 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991. doi:10.1145/114005.102808.

28 Maurice Herlihy and Jeannette Wing. Linearizability: a correcteness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

29 Udo Fritzke Jr., Philippe Ingels, Achour Mostéfaoui, and Michel Raynal. Consensus-based fault-tolerant total order multicast. *IEEE Trans. Parallel Distributed Syst.*, 12(2):147–156, 2001. doi:10.1109/71.910870.

30 Long Hoang Le, Mojtaba Eslahi-Kelorazi, Paulo R. Coelho, and Fernando Pedone. Ramcast: Rdma-based atomic multicast. In Kaiwen Zhang, Abdelouahed Gherbi, Nalini Venkatasubramanian, and Luís Veiga, editors, *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*, pages 172–184. ACM, 2021. doi:10.1145/3464298.3493393.

31 Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In Robert S. Swarz, Philip Koopman, and Michel Cukier, editors, *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012*, pages 1–12. IEEE Computer Society, 2012. doi:10.1109/DSN.2012.6263916.

32 Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 517–532. USENIX Association, 2016. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/mu.

33 Luís E. T. Rodrigues, Rachid Guerraoui, and André Schiper. Scalable atomic multicast. In *Proceedings of the International Conference On Computer Communications and Networks (ICCCN 1998), October 12-15, 1998, Lafayette, Louisiana, USA*, pages 840–847. IEEE Computer Society, 1998. doi:10.1109/ICCCN.1998.998851.

34 Nicolas Schiper and Fernando Pedone. Solving atomic multicast when groups crash. In Theodore P. Baker, Alain Bui, and Sébastien Tixeuil, editors, *Principles of Distributed Systems, 12th International Conference, OPODIS 2008, Luxor, Egypt, December 15-18, 2008. Proceedings*, volume 5401 of *Lecture Notes in Computer Science*, pages 481–495. Springer, 2008. doi:10.1007/978-3-540-92221-6_30.

35 Nicolas Schiper, Pierre Sutra, and Fernando Pedone. Genuine versus non-genuine atomic multicast protocols for wide area networks: An empirical study. In *28th IEEE Symposium on Reliable Distributed Systems (SRDS 2009), Niagara Falls, New York, USA, September 27-30, 2009*, pages 166–175. IEEE Computer Society, 2009. doi:10.1109/SRDS.2009.12.

36 Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010*, pages 214–224. IEEE Computer Society, 2010. doi:10.1109/SRDS.2010.32.

37 Pierre Sutra. The weakest failure detector for genuine atomic multicast (extended version), 2022. doi:10.48550/ARXIV.2208.07650.

38 Gadi Taubenfeld. Contention-sensitive data structures and algorithms. *Theoretical Computer Science*, 677:41–55, 2017. doi:10.1016/j.tcs.2017.03.017.