Optimal Atomic Broadcast and Multicast Algorithms for Wide Area Networks

Nicolas Schiper[†] Fernando Pedone[†]

[†]Faculty of Informatics
University of Lugano
6900 Lugano, Switzerland

University of Lugano
Faculty of Informatics
Technical Report No. 2007/004 Revision 1
August 2007

Abstract

In this paper, we study the atomic broadcast and multicast problems, two fundamental abstractions for building fault-tolerant systems. As opposed to atomic broadcast, atomic multicast allows messages to be addressed to a subset of the processes in the system, each message possibly being multicast to a different subset. We require atomic multicast algorithms to be *genuine*, i.e., only processes addressed by the multicast message are involved in the protocol. Our study focuses on wide area networks where *groups of processes*, i.e., processes physically close to each other, are inter-connected through high latency communication links. In this context, we capture the cost of algorithms, denoted *latency degree*, as the number of inter-group message delays between the broadcasting (multicasting) of a message and its delivery.

We present an atomic multicast algorithm with a latency degree of two and show that it is optimal. We then present the first fault-tolerant atomic broadcast algorithm with a latency degree of one. To achieve such a low latency, the algorithm is proactive, i.e., it may take actions even though no messages are broadcast. Nevertheless, it is quiescent: provided that the number of broadcast messages is finite, the algorithm eventually ceases its operation. As a consequence, in runs where the algorithm becomes quiescent too early, its latency degree is two. We show that this is unavoidable, and establish a lower bound on the quiescence of atomic broadcast algorithms. These two lower bound results stem from a common cause, namely the *reactiveness* of the processes at the time when the message is cast (broadcast or multicast). This reveals an interesting link between the quiescence of total order algorithms and the genuineness of atomic multicast, two problems which seemed to be unrelated at first sight.

1 Introduction

Distributed applications spanning multiple geographical locations have become common in recent years. Typically, each geographical site, or *group*, hosts an arbitrarily large number of processes connected through high-end local links; a few groups exist, interconnected through high-latency communication links. As a consequence, communication among processes in the same group is cheap and fast; communication among processes in different groups is expensive and orders of magnitude slower than local communication. Application data is replicated both locally, for high availability, and globally, usually for locality of access. In this paper we investigate the atomic broadcast and multicast problems, two communication primitives that offer adequate properties, namely agreement on the set of messages delivered and on their delivery order, to implement data replication [9].

Ideally, we would like to devise algorithms that use inter-group links as sparingly as possible, saving on both latency and bandwidth (i.e., number of messages). As we explain next, however, atomic broadcast and multicast establish an inherent tradeoff in this context. As opposed to atomic broadcast, atomic multicast allows messages to be sent to a subset of processes in the system. More precisely, messages can be addressed to any subset of the system's groups, each message possibly being multicast to a different subset. From a problem solvability point of view, atomic multicast can be easily reduced to atomic broadcast: every message is broadcast to all the groups in the system and only delivered by those processes the message is originally addressed to. Obviously, this solution is inefficient as it implies communication among processes that are not concerned by the multicast messages. To rule out trivial implementations of no practical interest, we require multicast algorithms to be *genuine* [7], i.e., only processes addressed by the message should be involved in the protocol. A genuine atomic multicast can thus be seen as an adequate communication primitive for distributed applications spanning multiple geographical locations in which processes store a subset of the application's data (i.e., *partial replication*).

To measure the latency cost of broadcast (multicast) algorithms, we use their *latency degree*. Informally—a precise definition is presented in Section 2—the latency degree is the minimum number of inter-group message delays between the cast of a message m and the last delivery of m among the processes that deliver m.

We show that for messages multicast to at least two groups, no genuine atomic multicast algorithm can hope to achieve a latency degree lower than two. This result is proven under strong system assumptions, namely processes do not crash and links are reliable. Moreover, this lower bound is tight, i.e., the fault-tolerant algorithm $\mathcal{A}1$ of Section 4 and the algorithm in [5] achieve this latency degree ($\mathcal{A}1$ is an optimized version of [5], see Section 4 for more details). A corollary of this result is that Skeen's algorithm, initially described in [2] and designed for failure-free systems, is also optimal—a result that has apparently been left unnoticed by the scientific community for more than 20 years.

We demonstrate that atomic multicast is inherently more expensive than atomic broadcast by presenting the first fault-tolerant broadcast algorithm with a latency degree of one. To achieve such a low latency, the algorithm is proactive, i.e., it may take actions even though no messages are broadcast. Nevertheless, we show how it can be made *quiescent*: provided that a finite number of messages is broadcast, processes eventually cease to communicate. In runs where the algorithm becomes quiescent too early, that is, a message m is broadcast after processes have decided to stop communicating, m will not be delivered in a single inter-group message delay, but in two. We show that this extra cost is unavoidable, i.e., no quiescent atomic broadcast algorithm can hope to always achieve a latency degree of one.[1]

These two lower bound results stem from a common cause, namely the *reactiveness* of the processes at the time when the message is cast. Roughly speaking, a process p is said to be *reactive* when the next

[1] This result also holds for quiescent (genuine or non-genuine) atomic multicast algorithms. The genuine case is already covered by the first lower bound result and is therefore irrelevant here.

message m that p sends is either multicast or sent in response to the reception of a message. In Section 3, we first show that no atomic broadcast or multicast algorithm can hope to deliver the last cast message m with a latency degree of one if m is cast at a time when processes are reactive. To obtain the lower bounds, we then show that (i) in runs of any genuine atomic multicast algorithm where one message is multicast at time t, processes are reactive at t and (ii) in runs of any quiescent atomic broadcast or atomic multicast algorithm where a finite number of messages are cast, processes are eventually reactive forever.

These results help better understand the difference between atomic broadcast and multicast. In particular, they point out a tradeoff between the latency degree and message complexity of these two problems. Consider a partial replication scenario where each group replicates a set of objects. If latency is the main concern, then every operation should be broadcast to all groups, and only groups concerned by the operation handle it. This solution, however, has a high message complexity: every operation leads to sending at least one message to all processes in the system. Obviously, this is inefficient if the operation only *touches* a subset of the system's groups. To reduce the message complexity, genuine multicast can be used. However, any genuine multicast algorithm will have a latency degree of at least two.

The rest of the paper is structured as follows. In Section 2, we present our system model and definitions. Section 3 shows the genuine atomic multicast latency degree lower bound and investigates the cost of quiescence in a unified way. In Sections 4 and 5, we present the optimal multicast and broadcast algorithms. Finally, Section 6 discusses the related work and concludes the paper. The proofs of correctness of the algorithms can be found in the appendix.

2 System Model and Definitions

2.1 Processes and Links

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. We assume the benign crash-stop failure model, i.e., processes may fail by crashing, but do not behave maliciously. A process that never crashes is $correct$; otherwise it is $faulty$. 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 or 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. Hereafter, we assume that in each group: (1) there exists at least one correct process and (2) consensus is solvable (the consensus problem is defined below).

2.2 Specifications of Agreement Problems

We define the four agreement problems considered in this paper, namely consensus, reliable multicast, atomic multicast, and atomic broadcast. Let $\mathcal{A}$ be an algorithm solving an agreement problem. We define $\mathcal{R}(\mathcal{A})$ as the set of all admissible runs of $\mathcal{A}$.

Consensus Throughout the paper, we assume the existence of a *uniform consensus* abstraction. In the *consensus* problem, processes propose values and must reach agreement on the value decided. Uniform consensus is defined by the primitives propose(v) and decide(v) and satisfies the following properties [8]: (i) *uniform integrity:* if a process decides v, then v was previously proposed by some process, (ii) *termination:* every correct process eventually decides exactly one value, (iii) *uniform agreement:* if a process decides v, then all correct processes eventually decide v.

Reliable Multicast Our algorithms also use a *non-uniform reliable multicast* primitive. As opposed to reliable broadcast [8], messages may be reliably multicast to a subset of the processes in Π. For each message m, $m.dest$ denotes the processes to which the message is reliably multicast. Non-uniform reliable multicast is defined by primitives ***R-MCast(m)*** and ***R-Deliver(m)***, and satisfies the following properties : (i) *uniform integrity:* for any process p and any message m, p R-Delivers m at most once, and only if $p \in m.dest$ and m was previously R-MCast, (ii) *validity:* if a correct process p R-MCasts a message m, then eventually all correct processes $q \in m.dest$ R-Deliver m, (iii) *agreement:* if a correct process p R-Delivers a message m, then eventually all correct processes $q \in m.dest$ R-Deliver m.

Atomic Multicast Atomic multicast is defined by the primitives A-MCast and A-Deliver. Atomic multicast allows messages to be A-MCast to a subset of groups in Γ. For each message m, $m.dest$ denotes the groups to which m is A-MCast. Let p be a process. By abuse of notation, we write $p \in m.dest$ instead of $\exists g \in \Gamma : g \in m.dest \wedge p \in g$. Hereafter, we denote the sequence of messages A-Delivered by p at time t as S_p^t, and the sequence of messages A-Delivered by p at time t *projected* on processes p and q as $P_{p,q}(S_p^t)$, i.e., $P_{p,q}(S_p^t)$ is the sequence of messages S_p^t restricted to the messages m such that $p, q \in m.dest$. Atomic multicast satisfies the uniform integrity and validity properties of reliable multicast as well as the two following properties: (i) *uniform agreement:* if a process p A-Delivers m, then all correct processes $q \in m.dest$ eventually A-Deliver m, (ii) *uniform prefix order:* for any two processes p and q and any time t, either $P_{p,q}(S_p^t)$ is a prefix of $P_{p,q}(S_q^t)$ or $P_{p,q}(S_q^t)$ is a prefix of $P_{p,q}(S_p^t)$.

We also require atomic multicast algorithms to be *genuine* [7]:

- *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 $p \in m.dest$.

Atomic Broadcast Atomic broadcast is a special case of atomic multicast. It is defined by the primitives A-BCast and A-Deliver and satisfies the same properties as atomic multicast where all A-BCast messages m are such that $m.dest = \Gamma$, i.e., messages are always A-BCast to all groups in the system.

2.3 Latency Degree

Let $\mathcal{A}$ be a broadcast or multicast algorithm and R be a run of $\mathcal{A}$ ($R \in \mathcal{R}(\mathcal{A})$). Moreover, in run R, let m be a message A-XCast (A-BCast or A-MCast) and $\Pi'(m) \subseteq \Pi$ be the set of processes that A-Deliver m. Intuitively, the latency degree of R is the minimal length of the causal path between the A-XCast of m and the last A-delivery of m among the processes in $\Pi'(m)$, when counting inter-group messages only. To define this latency degree we assign timestamps to process events using a slightly modified version of Lamport's logical clocks [9]. Initially, for all processes $p \in \Pi$, p's logical clock, LC_p, is initialized to 0. On process p, an event e is assigned its timestamp as follows:

1. If e is a local event, $ts(e) = LC_p$
2. If e is the send event of a message m to a process q,

$$ts(e) = \begin{cases} LC_p + 1, & if\ group(p) \neq group(q) \\ LC_p, & otherwise \end{cases}$$

3. If e is the receive event of a message m, $ts(e) = \max(LC_p, ts(send(m)))$

The latency degree of a message m A-XCast in run R is defined as follows:

$$\Delta(m, R) = \max_{q \in \Pi'(m)}(ts(A\text{-}Deliver(m)_q) - ts(A\text{-}XCast(m)_p))$$

where $A\text{-}Deliver(m)_q$ and $A\text{-}XCast(m)_p$ respectively denote the A-Deliver(m) event on process q and the A-XCast(m) event on process p. We refer to the latency degree of an algorithm $\mathcal{A}$ as the minimum value of $\Delta(m, R)$ among all admissible runs R of $\mathcal{A}$ and messages m A-XCast in R.[2]

3 The Inherent Cost of Reactiveness

We establish the inherent cost of the genuine atomic multicast problem for messages that are multicast to multiple groups and we show that quiescence has a cost, i.e., in runs where a message m is cast at a time when the algorithm is quiescent, there exists no algorithm that delivers m with a latency degree of one. As explained in Section 1, we proceed in two steps. We first show that, if processes are reactive when the last message m is cast, then m cannot be delivered with a latency degree of one. We then prove that (i) in runs of any genuine atomic multicast algorithm where one message is multicast at time t, processes are reactive at t and (ii) in runs of any quiescent atomic broadcast or atomic multicast algorithm where a finite number of messages are cast, processes are eventually reactive forever.

The proofs are done in a model identical to the model of Section 2, except that processes do not crash and links are reliable, i.e., they do not corrupt, duplicate, or loose messages.

Definition 3.1 *In a run R of an atomic broadcast or multicast algorithm, we say that a process p is reactive at time t iff p sends a message m at time $t' \geq t$ only if p A-XCasts m or if p received a message sent in the interval $[t, t']$.*

Proposition 3.1 *In a system with at least two groups, for any atomic broadcast or any atomic multicast algorithm $\mathcal{A}$, there does not exist runs R_1, R_2 of $\mathcal{A}$ in which processes are reactive at the time the last messages m_1, m_2 are A-XCast to at least two groups, such that $\Delta(m_1, R_1) = \Delta(m_2, R_2) = 1$.*

Proof: Suppose, by way of contradiction, that there exist an algorithm $\mathcal{A}$ and runs R_i of $\mathcal{A}$ ($i \in \{1, 2\}$) such that $\Delta(m_i, R_i) = 1$. Consider two groups, g_1 and g_2. In run R_i, process $p_i \in g_i$ A-XCasts message m_i at time t to g_1 and g_2. We first show that (*) in R_i, at or after time t, processes can only send messages m such that for a sequence of events $e_1 = \text{A-XCast}(m_i), e_2, ..., e_k = \text{send}(m)$, $\text{A-XCast}(m_i) \rightarrow e_2 \rightarrow ... \rightarrow \text{send}(m)$.[3] Suppose, by way of contradiction, that there exists a process p in R_i that sends a message m at a time $t'_i \geq t$ such that the event send(m) is not causally linked to the event A-XCast(m_i). We construct a run R'_i identical to run R_i except that message m_i is not A-MCast (note that processes are also reactive at time t in R'_i). Since in R_i, there is no causal chain linking the event A-XCast(m_i) with the event send(m), runs R'_i and R_i are indistinguishable to process p up to and including time t'_i. Therefore, p also sends m in R'_i. Hence, since processes are reactive at time t and no message is A-XCast at or after t, p must have received a messag m' sent at or after t by some process q. Applying the same reasoning multiple times, we argue that there must exist a process r that sends a message m'' at time t such that for some events $e_1 = \text{send}(m''), e_2, ..., e_{x-1} = \text{send}(m'), e_x = \text{send}(m)$, we have $\text{send}(m'') \rightarrow ... \rightarrow \text{send}(m') \rightarrow \text{send}(m)$. However, r cannot send m'' because no message is A-XCast at or after t, a contradiction.

By the validity property of $\mathcal{A}$ and because there is no failure, all processes eventually A-Deliver m_i. Since $\Delta(m_i, R_i) = 1$, by (*), processes in g_i A-Deliver m_i before receiving any message from processes in g_{3-i} sent at or after time t. Let $t^*_i > t$ be the time at which all processes in g_i have A-Delivered message

[2]Note that we also use the latency degree to capture the cost of reliable multicast and consensus. For these two agreement problems, the definition is identical, except that the event A-XCast is respectively replaced by R-MCast and propose, and the event A-Deliver is respectively replaced by R-Deliver and decide (in the case of consensus, m refers to the consensus instance number).

[3]Events $e_1, ..., e_k$ can be of four kinds, either send(m), receive(m), A-XCast(m), or A-Deliver(m) for some message m. Moreover, the relation $\rightarrow$ is Lamport's transitive happened before relation on events [9]. It 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 $e_1 = \text{send}(m)$ and $e_2 = \text{receive}(m)$ for some message m.

m_i. We now build run R_3 as follows. As in run R_i, p_i A-XCasts m_i. Runs R_i and R_3 are indistinguishable for processes in group g_i up to time t_i^*, that is, all messages causally linked to the event A-XCast(m_{3-i}) (including A-XCast(m_{3-i}) itself) sent from processes in group g_{3-i} to processes in group g_i are delayed until after t_i^*. Consequently, processes in group g_i have all A-Delivered m_i by time t_i^*. By the uniform agreement of $\mathcal{A}$, processes in g_1 eventually A-Deliver m_2 and processes in g_2 eventually A-Deliver m_1, violating the uniform prefix order property of $\mathcal{A}$. □

Proposition 3.2 *For any run R of any genuine atomic multicast algorithm $\mathcal{A}$ where one message is A-MCast at time t, processes are reactive at time t.*

Proof: In run R, by the genuineness property of $\mathcal{A}$, for any message m' sent, there exist events $e_1 =$ A-MCast(m), $e_2, ..., e_x =$ send(m') such that A-MCast(m) $\rightarrow e_2 \rightarrow ... \rightarrow$ send(m') (otherwise, using a similar argument as in Proposition 3.1, we could build a run R' identical to run R, except that no message is A-MCast in R', such that a process sends a message anyway, contradicting the fact that in R' no message is A-MCast and $\mathcal{A}$ is genuine).

Consequently, for any process p, if p sends a message m' at $t' \geq t$, then p A-MCasts m' or p received a message in the interval $[t, t']$. □

Proposition 3.3 *For any run R of any quiescent atomic broadcast or atomic multicast algorithm $\mathcal{A}$ in which a finite number of messages are A-XCast, there exists a time t such that for all $t' \geq t$, processes are reactive at t'.*

Proof: In R, a finite number of messages are A-XCast. Because $\mathcal{A}$ is quiescent, there exists a time t at or after which no messages are sent. It follows directly that for all $t' \geq t$ processes are reactive at t'. □

Although our result shows that if the last message m is cast when processes are reactive, then m cannot be delivered in one inter-group message delay, in practice, multiple messages may bear this overhead. In fact, this might even be the case in runs where an infinite number of messages are cast. Indeed, to ensure quiescence, processes must, in some way or another, *predict* whether any message will be cast in the future. Hence, if the prediction is negative, processes must then eventually stop communicating, and this may be prematurate.

4 Atomic Multicast for WANs

In this section, we present a latency degree-optimal atomic multicast algorithm which is inspired by the one from Fritzke *et al.* [5], an adaptation of Skeen's algorithm for failure-prone systems. We first explain the basic principle of our algorithm and how it differs from [5]. We then explain our algorithm in detail.

4.1 Algorithm Overview

The algorithm associates every multicast message with a timestamp. To ensure 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. To satisfy these two properties, inside each group g, processes implement a logical *clock* that is used to generate timestamps, this is g's clock. To guarantee g's clock consistency, processes use consensus to maintain it. Moreover, every message m goes trough the following four stages:

- *Stage* s_0: In every group $g \in m.dest$, processes define a timestamp for m using g's clock. This is g's proposal for m's final timestamp.

- *Stage* s_1: Groups in $m.dest$ exchange their proposals for m's timestamp and set m's final timestamp to the maximum timestamp among all proposals.
- *Stage* s_2: Every group in $m.dest$ sets its clock to a value greater than the final timestamp of m.
- *Stage* s_3: Message m is A-Delivered when its timestamp is the smallest among all messages that are in one of the four stages and not yet A-Delivered.

As mentioned above, our algorithm differentiates itself from [5] in several aspects. First, when a message is multicast, instead of using a uniform reliable multicast primitive, we use a non-uniform version of this primitive while still ensuring properties as strong as in [5]. Second, in contrast to [5], not all messages go trough all four stages. For messages that are multicast to only one group, our algorithm allows them to *jump* from stage s_0 to stage s_3 directly. Also, even for messages that are multicast to more than one group, on processes belonging to a group that has proposed a timestamp equal to the final timestamp of m (the biggest proposal of all), m skips stage s_2.

4.2 The Algorithm in Detail

Algorithm $\mathcal{A}1$ is composed of two concurrent tasks. Each line of the algorithm is executed atomically. Apart from application data, messages are composed of four fields: $dest$, id, ts, and $stage$. For every message m, $m.dest$, indicates to which group 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. On every process p, four global variables are used: K is p's copy of $group(p)$'s clock and also denotes the current consensus instance in execution or the next to be executed, $propK$ forbids p to propose more than one value per consensus instance, $PENDING$ is the set of messages that have not yet been A-Delivered, and $ADELIVERED$ is the set of A-Delivered messages. We explain Algorithm $\mathcal{A}1$ by describing the actions a process p takes when a message m is in one of the four possible stages.

Stage s_0: For p to A-MCast m, p R-MCasts m to processes in $m.dest$. When p R-Delivers m, if m has not been added to $PENDING$ at line 30 or A-Delivered before, p sets $m.stage$ to s_0 and adds m to the $PENDING$ set. Note that p also sets m's timestamp to the current value of K to guarantee that every message in $PENDING$ is associated with a timestamp. In order for processes in each group of $m.dest$ to agree on their timestamp proposal, a consensus instance inside each group is executed. Hence, p checks that $propK \leq K$ (line 14) to verify that no consensus instance is currently running and, if it is the case, p proposes m to the next consensus instance. Process p actually proposes all messages in $PENDING$ that are either in stage s_0 or s_2 to share the cost of consensus instances among the set of messages proposed and to allow messages in different stages to make progress in parallel. As soon as a consensus instance k decides on m ($m \in msgSet'$ at line 18), p takes the following actions. First, if m is A-MCast to more than one group, then m transitions to stage s_1 and $group(p)$'s proposal for m's timestamp is k (lines 22-23). Otherwise, if m is A-MCast to one group only, m's final timestamp is k and m transitions to stage s_3 directly (lines 28-29). Indeed, at this point in time, processes in $m.dest$ already agree on m's timestamp because p's group is the only group in $m.dest$. Moreover, p's copy of $group(p)$'s clock, K, will be greater than $m.ts$ after p executes line 31.

Stage s_1: Process p sends its group's proposal to all the groups in $m.dest$ different from p's group (line 24).[4] Once p receives all the required timestamps' proposals (line 33), p gathers them in a set called $TSset$ and computes the maximum value, max, of that set. If the proposal of p's group is bigger or equal

[4]Note that this message also serves the purpose of propagating m. Indeed, consider a scenario where the process that A-MCasts m is faulty and m is A-MCast to multiple groups. Because the reliable multicast primitive we use is non-uniform, it is possible that only faulty processes in a group g R-Deliver m. After processes in g decide on m in consensus and send the (TS, m) message at line 24, as there is at least one correct process per group, we guarantee that correct processes in $m.dest$ receive m. Hence, each group in $m.dest$ eventually defines its timestamp proposal and m reaches stage s_3 on all correct processes in $m.dest$, thus ensuring liveness of the algorithm.

to max, then m can skip stage s_2. Indeed, at this point in time, p's copy of $group(p)$'s clock, K, is already bigger than $m.ts$, because p previously executed line 31. On the other hand, if the proposal of p's group is smaller than max, p sets m's timestamp to max and m transitions to stage s_2 (line 39-40).

Stage s_2: Process p then keeps on proposing m to consensus instances until an instance k decides on m. When instance k terminates, m transitions to stage s_3 at line 26 and p sets K to a value greater than m's final timestamp at line 31.

Stage s_3: After m reaches stage s_3 (at lines 26, 29, or 36), p checks whether m can be A-Delivered by executing the procedure ADeliveryTest. This procedure A-Delivers m only if m has the smallest timestamp among all messages in $PENDING$ (line 4). If two messages m_1 and m_2 have the same timestamp, we break ties using their message identifier. More precisely, $(m_1.ts, m_1.id) < (m_2.ts, m_2.id)$ is true either if $m_1.ts < m_2.ts$ or if $m_1.ts = m_2.ts$ and $m_1.id < m_2.id$.

4.3 Latency Degree Analysis

Consider a message m that is multicast by a process p. If m is multicast to one group, it is easy to see from Algorithm $\mathcal{A}1$ that the latency degree is zero if $p \in g$, and one otherwise. This is obviously optimal. In the case where m is multicast to multiple groups, we show in the appendix that the latency degree is two, which matches the lower bound of Section 3.

Theorem 4.1 *There exists a run R of algorithm $\mathcal{A}1$ in which a message m is A-MCast to two groups such that $\Delta(m, R) = 2$.*

5 Atomic Broadcast for WANs

In this section, we present the first fault-tolerant atomic broadcast algorithm that achieves a latency degree of one. Together with the lower bound of Section 3, this shows that atomic multicast is more costly than atomic broadcast. We present the main idea of this algorithm, explain it in detail, and conclude with a discussion on its latency degree.

5.1 Algorithm Overview

To atomically broadcast a message m, a process p reliably multicasts m to the processes in p's group. In parallel, processes execute an *unbounded* sequence of rounds. At the end of each round, processes deliver a set of messages according to some deterministic order. To ensure agreement on the messages 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 delivered at the end of round r is the union of all bundles. Note that we also wish to ensure *quiescence*, i.e., if there is a time after which no message is broadcast, then processes eventually stop sending messages. To do so, processes try to predict when no further messages will be broadcast. When the algorithm predicts that messages will no longer be broadcast, processes stop executing rounds. Our algorithm is indulgent with regards to prediction mistakes, i.e., if processes become quiescent too early, they can restart so that liveness is still ensured. We explain in the section below how this is done.

5.2 The Algorithm in Detail

Algorithm $\mathcal{A}2$ is composed of four concurrent tasks. Each line of the algorithm is executed atomically. On every process p, six global variables are used: K denotes the current round number, $propK$ forbids p to propose more than one value per consensus instance, $RDELIVERED$ and $ADELIVERED$ are the set of

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

1: **Initialization**
2: $K \leftarrow 1$, $propK \leftarrow 1$, $PENDING \leftarrow \emptyset$, $ADELIVERED \leftarrow \emptyset$

3: **procedure** ADeliveryTest()
4: **while** $\exists m \in PENDING : m.stage = s_3 \wedge$
$\forall m' \in PENDING : m' \neq m \Rightarrow (m.ts, m.id) < (m'.ts, m'.id)$ **do**
5: A-Deliver(m)
6: $ADELIVERED \leftarrow ADELIVERED \cup \{m\}$
7: $PENDING \leftarrow PENDING \setminus \{m\}$

8: **To A-MCast** message m {*Task 1*}
9: R-MCast(m) to $\{q \mid q \in m.dest\}$

10: **When** (R-Deliver(m) $\vee$ receive(TS, m)) $\wedge$
$m \notin PENDING \cup ADELIVERED$ {*Task 2*}
11: $m.ts \leftarrow K$
12: $m.stage \leftarrow s_0$
13: $PENDING \leftarrow PENDING \cup \{m\}$

14: **When** ($\exists m \in PENDING : m.stage = s_0 \ \vee \ m.stage = s_2$) $\wedge$
$propK \leq K$
15: $msgSet = \{m \mid m \in PENDING \wedge (m.stage = s_0 \vee m.stage = s_2)\}$
16: Propose(K, $msgSet$) ▷ consensus inside group
17: $propK \leftarrow K + 1$

18: **When** Decided(K, $msgSet'$)
19: **foreach** $m' \in msgSet'$ **do**
20: **if** $|m'.dest| > 1$ **then**
21: **if** $m'.stage = s_0$ **then**
22: $m'.ts \leftarrow K$
23: $m'.stage \leftarrow s_1$
24: send(TS, m') to $\{q \mid q \in m.dest \wedge group(q) \neq group(p)\}$
25: **else**
26: $m'.stage \leftarrow s_3$
27: **else**
28: $m'.ts \leftarrow K$
29: $m'.stage \leftarrow s_3$ ▷ second consensus not needed
30: $PENDING \leftarrow PENDING \cup \{msgSet'\}$ ▷ add message or update its fields
31: $K \leftarrow \max(\max_{m' \in msgSet'}(m'.ts), K) + 1$
32: ADeliveryTest()

33: **When** $\exists m \in PENDING : m.stage = s_1 \ \wedge$
$\forall g \in (m.dest \setminus group(p)) \ \exists q \in g$: received(TS, m) from q
34: $TSset = \{m.ts \mid \exists q \in m.dest$: received (TS, m) from $q\}$
35: **if** $m.ts \geq \max_{ts \in TSset}(ts)$ **then**
36: $m.stage \leftarrow s_3$ ▷ second consensus not needed
37: ADeliveryTest()
38: **else**
39: $m.ts \leftarrow \max_{ts \in TSset}(ts)$
40: $m.stage \leftarrow s_2$

R-Delivered and A-Delivered messages respectively, $Msgs$ is used to store the groups' message bundles, and $Barrier$ denotes the last round p currently thinks it will execute.

To A-BCast a message m, a process p R-MCasts m to p's group (line 5). When p R-Delivers m, p adds it to $RDELIVERED$. At the beginning of every round r, p proposes, in the next consensus instance, the

set of messages that have been R-Delivered but not A-Delivered yet (line 12). Note that this proposal may be the empty set. When this instance decides on a set of messages $msgSet'$ (line 14), p's group message bundle in round r, p sends $msgSet'$ to all the groups different from $group(p)$. Process p then waits to receive the message bundles of round r from all the other groups and A-Delivers the union of all bundles in some deterministic order (lines 16-20).

In order to ensure quiescence, processes try to predict when no message will be broadcast anymore. Our prediction strategy is simple. If no message was A-Delivered in the current round (line 22), p leaves $Barrier$ untouched, in which case p will not execute the next round. This follows from the fact that K is incremented by one at the end of each round (line 21), and if no message is A-BCast anymore and all R-Delivered messages were A-Delivered, line 11 never evaluates to true anymore and p does not execute further rounds.

To tolerate prediction mistakes, the algorithm proceeds as follows. Consider a process p that broadcasts a message m after all processes have become quiescent. Let r be the last round processes executed. After processes in p's group R-Deliver m, line 11 evaluates to true and they start executing round $r + 1$. To allow processes in other groups to restart executing rounds as well, after receiving p's group message bundle (line 8), these processes set $Barrier$ to $r + 1$. Hence, line 11 evaluates to true and they start round $r + 1$.

Algorithm $\mathcal{A}2$ Atomic Broadcast - Code of process p

```
 1: Initialization
 2:    K ← 1, propK ← 1, RDELIVERED ← ∅, ADELIVERED ← ∅
 3:    Msgs ← ∅, Barrier ← 0

 4: To A-BCast message m                                              {Task 1}
 5:    R-MCast m to {q | group(q) = group(p)}

 6: When R-Deliver(m)                                                 {Task 2}
 7:    RDELIVERED ← RDELIVERED ∪ {m}

 8: When receive(x, msgSet) from q                                    {Task 3}
 9:    Msgs ← Msgs ∪ (x, q, msgSet)
10:    Barrier ← max(Barrier, x)

11: When ((RDELIVERED \ ADELIVERED) ≠ ∅ ∨
          K ≤ Barrier) ∧ propK ≤ K                                    {Task 4}
12:    Propose(K, RDELIVERED \ ADELIVERED)          ▷ consensus inside group
13:    propK ← K + 1

14: When Decided(K, msgSet')
15:    send(K, msgSet') to {q | q ∈ Π ∧ group(q) ≠ group(p)}
16:    wait until ∀g ∈ (Γ \ group(p)) : ∃q ∈ g s. t. received (K, -) from q
17:    Msgs ← Msgs ∪ (K, p, msgSet')
18:    msgsToADel ← {m | (K,-, msgSet) ∈ Msgs ∧ m ∈ msgSet}
19:    A-Deliver messages in msgsToADel in some deterministic order
20:    ADELIVERED ← ADELIVERED ∪ msgsToADel
21:    K ← K + 1
22:    if msgsToADel ≠ ∅ then                          ▷ stop executing rounds?
23:       Barrier ← max(Barrier, K)
```

5.3 Latency Degree Analysis

In the appendix, we analyze the latency degree of algorithm $\mathcal{A}2$. We first show that its best latency degree (among all its admissible runs) is one, which is optimal. We then consider runs where processes become

quiescent too early, i.e., processes stop executing rounds before a message is broadcast. In these runs, the latency degree of the algorithm is two.

Theorem 5.1 *There exists a run R of algorithm $\mathcal{A}2$ in which a message m is A-BCast such that $\Delta(m, R) = 1$.*

Theorem 5.2 *There exists a run R of algorithm $\mathcal{A}2$ in which the last message m is A-BCast when processes are reactive such that $\Delta(m, R) = 2$.*

It is worth noting that the presented broadcast algorithm never becomes reactive if the time between two consecutive broadcasts is smaller than the time to execute a round. Moreover, in this case, all rounds are *useful*, i.e., they all deliver at least one message, a scenario which can be considered as optimal. In a large-scale system where the inter-group latency is 100 milliseconds, a broadcast frequency of 10 messages per second is sufficient for the algorithm to reach this optimality. In case the broadcast frequency is too low or not constant, to prevent processes from stopping prematurely, more elaborate prediction strategies based on application behavior could be used.

6 Related Work and Final Remarks

The literature on atomic broadcast and multicast algorithms is abundant [3]. We here review the most relevant papers to our protocols.

Atomic Multicast In [7], the authors show the impossibility of solving genuine atomic multicast with unreliable failure detectors if groups are allowed to intersect. Hence, the algorithms cited below circumvent this impossibility result by considering non-intersecting groups that contain a sufficient number of correct processes to solve consensus. They can be viewed as variations of Skeen's algorithm [2], a multicast algorithm designed for failure-free systems, where messages are associated with timestamps and the message delivery follows the timestamp order. In [10], the addresses of a message m, i.e., the processes to which m is multicast, associate m with a timestamp. Processes then exchange their timestamps, 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 addresses of a message and can thus span multiple groups, this algorithm is not well-suited for wide area networks. In [4], 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 avoid cycles in the message delivery order, before handling other messages, every group waits for a final acknowledgment from group g_k. The latency degree of this algorithm is therefore proportional to the number of destination groups. In [5], as explained in Section 4, to ensure that processes agree on the timestamps associated to every message and to deliver messages according to the timestamp order, every message goes through four stages. In contrast to [5], the algorithm presented in this paper allows messages to skip stages, therefore sparing the execution of consensus instances. This has no impact on the latency degree or on the number of inter-group message sent as consensus instances are run inside groups. However, our algorithm sends fewer intra-group messages.

Atomic Broadcast In [1], the authors consider the atomic broadcast and multicast problems in a publish-subscribe system where links are reliable, publishers *do not crash*, and cast infinitely many messages. Agreement on the message ordering is ensured by using the same deterministic merge function at every subscriber process. Given the cast rate of publishers, the authors give optimal algorithms with regards to the merge delay, i.e., the time elapsed between the reception of a message by a subscriber and its delivery. Both

algorithms achieve a latency degree of one.[5] In [12], a time-based protocol is introduced to increase the probability of *spontaneous* total order in wide area networks by artificially delaying messages. Although the latency degree of the *optimistic* delivery of a message is one, the latency degree of its *final* delivery is two. Moreover, their protocol is non-uniform, i.e., the *agreement* property of Section 2 is only ensured for correct processes. In [13], a uniform protocol based on multiple sequencers is proposed. Every process p is assigned a sequencer that associates sequence numbers to the messages p broadcasts. Processes optimistically deliver a message m when they receive m's sequence number. The final delivery of m occurs when the sequence number of m has been validated by a majority of processes. The latency degree of this algorithm is identical to [12].

In Figure 1, we compare the latency degree and the number of inter-group exchanged messages of the aforementioned algorithms. In this comparison, we consider the best-case scenario, in particular there is no failure nor failure suspicion. We denote n as the total number of processes in the system, d as the number of processes in each group, and k as the number of groups to which a message is cast ($k \geq 2$). To compute the latency degree and number of inter-group message sent, we consider the oracle-based uniform reliable broadcast and uniform consensus algorithms of [6] and [11] respectively (note that [6] can easily be modified to implement reliable multicast). The latency degrees of [6] and [11] are respectively one and two. Furthermore, considering that a process p multicasts a message to k groups (we consider that p belongs to one of these k groups) or that k groups execute consensus, the algorithms respectively send $d(k-1)$ and $2kd(kd-1)$ inter-group messages.

Algorithm	latency degree	inter-group msgs.
[4]	$k+1$	$O(kd^2)$
[10]	4	$O(k^2d^2)$
[5]	2	$O(k^2d^2)$
Algorithm $\mathcal{A}1$	2	$O(k^2d^2)$
[1][6]	1	$O(kd)$

(a) Atomic Multicast

Algorithm	latency degree	inter-group msgs.
[12][7]	2	$O(n)$
[13]	2	$O(n^2)$
Algorithm $\mathcal{A}2$	1	$O(n^2)$
[1][6]	1	$O(n)$

(b) Atomic Broadcast

Figure 1: Comparison of the algorithms (d : nb. of processes per group, k : nb. of destination groups)

From Figure 1, we conclude that, among uniform fault-tolerant broadcast protocols, Algorithm $\mathcal{A}2$ achieves the best latency degree and message complexity. In the case of the atomic multicast problem, although Algorithm $\mathcal{A}1$ and [5] achieve the best latency degree among fault-tolerant protocols, [4] has a lower message complexity. Deciding which algorithm is best is not straightforward as it depends on factors such as the network topology as well as the latencies and bandwidths of links.

References

[1] M. K. Aguilera and R. E. Strom. Efficient atomic broadcast using deterministic merge. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 209–218, New York, NY, USA, 2000. ACM Press.

[2] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

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

[4] C. Delporte-Gallet and H. Fauconnier. Fault-tolerant genuine atomic multicast to multiple groups. In *OPODIS*, pages 107–122, 2000.

[5] Note that this does not contradict the latency degree lower bound of genuine atomic multicast. Indeed, their assumptions are different than ours, i.e., to ensure liveness of their multicast algorithm, they require that each publisher multicast infinitely many messages to *each* subscriber.

[6] This paper considers a strong model where links are reliable, *multicaster* processes do not crash, and multicast infinitely many messages to every process.

[7] This algorithm is non-uniform, i.e., it guarantees the agreement property of Section 2 only for correct processes.

[5] U. Fritzke, Ph. Ingels, A. Mostéfaoui, and M. Raynal. Fault-tolerant total order multicast to asynchronous groups. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 578–585, October 1998.

[6] S. Frolund and F. Pedone. Ruminations on domain-based reliable broadcast. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*, pages 148–162, London, UK, 2002. Springer-Verlag.

[7] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, 254(1-2):297–316, 2001.

[8] 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.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] L. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *Proceedings of the 7th IEEE International Conference on Computer Communications and Networks (IC3N'98)*, pages 840–847, Lafayette, Louisiana, USA, 1998.

[11] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[12] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic total order in wide area networks. In *Proc. 21st IEEE Symposium on Reliable Distributed Systems*, pages 190–199. IEEE CS, October 2002.

[13] P. Vicente and L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, page 92, Washington, DC, USA, 2002. IEEE Computer Society.

A Appendix

A.1 Latency Degree Analysis - Atomic Multicast

Theorem 4.1 *There exists a run R of algorithm $\mathcal{A}1$ in which a message m is A-MCast to two groups such that $\Delta(m, R) = 2$.*

Proof: We illustrate such a run R. Consider that there are two groups in the system: g_1 and g_2. At time t_1, process $p_1 \in g_1$ A-MCasts message m to g_1 and g_2. By time t_2 and t_3 respectively, processes in g_i ($i \in \{1, 2\}$) have R-Delivered m and have decided on m in consensus instance 1. Note that for all processes p, because the latency degree of non-uniform reliable multicast is one, the decide events dec_p is such that $ts(dec_p) = 1$. At time t_4, processes in g_i send(TS, m) to g_{3-i}. At time t_5, processes in g_i receive (TS, m). On all processes, line 35 evaluates to true and they A-Deliver m such that ts(A-Deliver(m)) = 2. □

A.2 Latency Degree Analysis - Atomic Broadcast

Theorem 5.1 *There exists a run R of algorithm $\mathcal{A}2$ in which a message m is A-BCast such that $\Delta(m, R) = 1$.*

Proof: We illustrate such a run R. Consider that there are two groups in the system: g_1 and g_2. Let r be a round where some message was A-Delivered. Hence, all processes start round $r + 1$. At time t_1, process $p_1 \in g_1$ A-BCasts message m_1. By time t_2 processes in g_1 have R-Delivered m_1. By time t_3 all processes have decided in consensus instance $r + 1$. At time t_4, processes in g_1 send a message ($r + 1$, $\{m\}$) to processes in g_2; processes in g_2 send a message ($r + 1$, $\emptyset$) to processes in g_1. At time t_5, processes in g_1 receive the message ($r + 1$, -) from g_2 and vice-versa. Hence, processes A-Deliver m such that ts(A-Deliver(m)) = 1. □

Theorem 5.2 *There exists a run R of algorithm $\mathcal{A}2$ in which the last message m is A-BCast when processes are reactive such that $\Delta(m, R) = 2$.*

Proof: We illustrate such a run R. Consider that there are two groups in the system: g_1 and g_2. At time t_1, process $p \in g_1$ A-BCasts message m. By time t_2 and t_3 respectively, processes in g_1 have R-Delivered m and have decided on m in consensus instance r. At time t_4 and t_5 respectively, processes in g_1 send(r, $\{m\}$) to g_2 and processes in g_2 receive this message. At time t_6, t_7, and t_8, processes in g_2 respectively decide in consensus instance r, send(r, $\emptyset$) to processes in g_1, and A-Deliver m. At time t_8, processes in g_1 A-Deliver m such that ts(A-Deliver(m)) = 2. □

A.3 The Algorithms' Proofs

Note that in the following proofs, for brevity, we often write: "process p decides on m in instance k" instead of writing "process p decides on $msgSet'$ such that $m \in msgSet'$ in instance k". We use the following definition of *is a prefix of*: S_1 is a prefix of $S_2 \Leftrightarrow \exists \alpha : S_1 \oplus \alpha = S_2$.

A.3.1 The Proof of Algorithm $\mathcal{A}1$

Definition A.1 *We denote as κ_p^t the sequence of values taken by variable K on process p up to time t.*

Lemma A.1 *For any two processes p, q such that $group(p) = group(q)$ and any time t, either κ_p^t is a prefix of κ_q^t or κ_q^t is a prefix of κ_p^t.*

Proof: We proceed by induction on the length l of κ_p^t.

- Base step ($l = 1$) : K is initialized to 1, therefore $\kappa_p^t = \{1\}$ and 1 is the first element of κ_q^t. Therefore, κ_p^t is a prefix of κ_q^t.
- Induction step: Suppose that Lemma A.1 holds for $x = l - 1$, we prove that Lemma A.1 holds for $x = l$. We do so by showing that $\neg(\kappa_p^t$ is a prefix of $\kappa_q^t) \Rightarrow \kappa_q^t$ is a prefix of κ_p^t. Suppose, by way of contradiction, that (*) $\neg(\kappa_p^t$ is a prefix of $\kappa_q^t) \wedge \neg(\kappa_q^t$ is a prefix of $\kappa_p^t)$. By the induction hypothesis, either (a) $\exists\alpha : \kappa_{p_{l-1}}^t \oplus \alpha = \kappa_q^t$ or (b) $\exists\beta : \kappa_q^t \oplus \beta = \kappa_{p_{l-1}}^t$.[8] We now show that (a) and (b) lead to a contradiction.
  - In case (a), $\kappa_{p_{l-1}}^t = \{k_1, .., k_{l-1}\}$, $\kappa_p^t = \{k_1, .., k_l\}$, and $\kappa_q^t = \{k_1, .., k_{l-1}\} \oplus \alpha'$. There are two cases to consider, (a-i) $\alpha' = \epsilon$ or (a-ii) $\alpha' \neq \epsilon$.
    - In case (a-i), $\alpha' = \epsilon$ and therefore κ_q^t is a prefix of κ_p^t, a contradiction to (*).
    - In case (a-ii), because $\neg(\kappa_p^t$ is a prefix of $\kappa_q^t)$, (**) the first integer k_a in α' is different from k_l. By the uniform agreement property of consensus, p and q decide on the same set of messages in instance k_{l-1}. Therefore, p and q set their variable K to the same value at line 31 when $K_p = K_q = k_{l-1}$. Consequently, $k_a = k_l$, a contradiction to (**).
  - In case (b), $\exists\beta : \kappa_q^t \oplus \beta = \kappa_{p_{l-1}}^t$ and therefore $\exists\beta' : \kappa_q^t \oplus \beta' = \kappa_p^t$, a contradiction to (*).

Lemma A.2 *For any correct process p, any t, and any process q such that $group(p) = group(q)$, there exists a t' such that $\kappa_p^{t'} = \kappa_q^t$.*

Proof: By Lemma A.1, either (a) κ_p^t is a prefix of κ_q^t or (b) κ_q^t is a prefix of κ_p^t.

- In case (a), κ_p^t is a prefix of κ_q^t. Therefore, there exists α such that $\kappa_p^t \oplus \alpha = \kappa_q^t$. Let k be the length of α and α_x be the prefix of α of length x. We show by induction on x that for $1 \leq x \leq k$, there exists a time t' such that $\kappa_p^{t'} = \kappa_p^t \oplus \alpha_x$.
  - Base step ($x = 1$): Let k_1 and k_2 be the last and only element of κ_p^t and α_1 respectively. Because there is a time at which $K_q = k_2$, q decided in instance k_1. By the uniform agreement property of consensus p eventually decides in instance k_1 and p, q decide on the same set of messages in that instance. Therefore, p eventually executes line 31 and sets K_p to k_2.
  - Induction step: Suppose that there exists a time t' such that $\kappa_p^{t'} = \kappa_p^t \oplus \alpha_{x-1}$, we show that this also holds for x ($1 \leq x \leq k$). The same argument as in the base step is used, where k_1 is the last element of α_{x-1} and k_2 is the last element of α_x.
- In case (b), κ_q^t is a prefix of κ_p^t, therefore there exists a time t' such that $\kappa_p^{t'} = \kappa_q^t$. □

Lemma A.3 *For any message m and any process p, after p adds m to $PENDING_p$ at line 13 or line 30, $m \in PENDING_p \cup ADELIVERED_p$ forever.*

Proof: Before m is removed from $PENDING_p$ at line 7, m is added to $ADELIVERED_p$ at line 6. Therefore, after m is added to $PENDING_p$ either at line 13 or line 30, $m \in PENDING_q \cup ADELIVERED_q$ forever. □

[8] $\kappa_{p_{l-1}}^t$ denotes the prefix of κ_p^t of length $l - 1$.

Lemma A.4 *For any message m and any process p:*

- *(a) p executes at most once line 18 when $m.stage = s_0$ with $m \in msgSet'$*
- *(b) p executes at most once line 18 when $m.stage = s_2$ with $m \in msgSet'$*

Proof:

- (a) Suppose, by way of contradiction, that p executes line 18 such that $m \in msgSet' \wedge m.stage = s_0$ more than once. Let k and k' ($k < k'$) be the first and second consensus instances such that p decides on a $msgSet'$ with $m \in msgSet' \wedge m.stage = s_0$ at line 18. By the uniform integrity property of consensus, there exists a process $q \in group(p)$ such that q proposes m to instance k' with $m.stage = s_0$. By Lemma A.1, q decides in instance k before proposing m in instance k'. By the uniform agreement property of consensus, q decides on m in instance k. By Lemma A.3, after q finishes executing line 30 when $K_q = k$, $m \in PENDING_q \cup ADELIVERED_q$ forever. Hence, after deciding in instance k, q cannot execute line 12 to set m's stage back to s_0. Therefore, q does not propose m to consensus instance k' such that $m.stage = s_0$, a contradiction.
- (b) Notice that p can only execute line 18 such that $m.stage = s_2$ with $m \in msgSet'$ if $|m.dest| > 1$. Suppose, by way of contradiction, that p executes line 18 such that $m \in msgSet' \wedge m.stage = s_2$ more than once. Let k_1 and k_3 ($k_1 < k_3$) be the first and second consensus instances such that p decides on m with $m.stage = s_2$. By the uniform integrity property of consensus, there exists a process $q \in group(p)$ such that q proposes m to instance k_3 with $m.stage = s_2$. By Lemma A.1, q decides in instance k_1 before proposing m in instance k_3. By the uniform agreement property of consensus, q decides on m in instance k_1 such that $m.stage = s_2$. Because q sets m's stage to s_3 after deciding in instance k_1 at line 26, after deciding in instance k_1 and before proposing m in instance k_3, either (b-i) q sets m's stage back to s_0 at line 12 or (b-ii) q sets m's stage back to stage s_1 at line 22. We show that (b-i) and (b-ii) lead to a contradiction.
  - In case (b-i), by Lemma A.3, after q executes line 30 when $K_q = k_1$, $m \in PENDING_q \cup ADELIVERED_q$ forever. Therefore, q does not execute line 12 after deciding in instance k_1, a contradiction.
  - In case (b-ii), there exists a consensus instance k_2 ($k_1 < k_2 < k_3$) such that q decides on m with $m.stage = s_0$ in instance k_2. By the uniform integrity property of consensus, there exists a process $r \in group(q)$ such that r proposes m in instance k_2. By Lemma A.1, r decides in instance k_1 before proposing m in instance k_2. By the uniform agreement property of consensus, r decides on m with $m.stage = s_2$ in instance k_1. By Lemma A.3, after r executes line 30 when $K_r = k_1$, $m \in PENDING_r \cup ADELIVERED_r$ forever. Therefore, r does not set m's stage back to s_0 at line 12 after deciding in instance k_1. Consequently, r does not propose m in instance k_2 such that $m.stage = s_0$, a contradiction. □

Lemma A.5 *For any message m and any process p, on p m transitions only once to stage s_3.*

Proof: There are two cases to consider:

- (1) $|m.dest| = 1$: Follows directly from Lemma A.4.
- (2) $|m.dest| > 1$: Message m transitions to stage s_3 either (i) at line 26 or (ii) at line 36. In case (i), by Lemma A.4, p decides on m such that $m.stage = s_2$ only once. Therefore, m transitions to stage s_3 at line 26 only once. In case (ii), by Lemma A.4, p decides on m such that $m.stage = s_0$ only once. Therefore, m transitions to stage s_1 at line 23 only once and consequently m transitions to stage s_3 at line 36 only once. □

Definition A.2 *From Lemma A.5 and because the timestamp of a message m does not change after m transitions to stage s_3, we can define $m.ts_p^{s_3}$ as the timestamp of m on a process p when $m.stage = s_3$.*

Proposition A.1 (Uniform Integrity) *For any process p and any message m, (a) p A-Delivers m at most once, and (b) only if $p \in m.dest$ and m was previously A-MCast.*

Proof:

- (a) By Lemma A.5, on p, m transitions to stage s_3 only once. Because m is only A-Delivered if $m.stage = s_3$ such that $m \in PENDING$ and because m is removed from $PENDING$ just after it has been A-Delivered, p A-Delivers m at most once.
- (b) Follows directly from the algorithm. □

Lemma A.6 *For any message m and any correct process p, if there exists a time at which $m \in PENDING_p$ such that $m.stage = s_0$, then for all correct processes $q \in group(p)$ there exists a time at which $m \in PENDING_q$.*

Proof: Process p can only add m to $PENDING_p$ such that $m.stage = s_0$ at line 13. Therefore, either m was R-Delivered or (TS, m) was received. In the first case, because p and q are correct and by the agreement property of reliable multicast, all correct processes $q \in group(p)$ eventually R-Deliver m. In the second case, because p and q are correct and links are quasi-reliable, all correct processes $q \in group(p)$ eventually receive (TS, m). Therefore q eventually adds m to $PENDING_q$ if $m \notin PENDING_q \cup ADELIVERED_q$. Note that if $m \in ADELIVERED_q$, there exists a time at which $m \in PENDING_q$. □

Lemma A.7 *For any message m and any correct process p:*

- *(a) if there exists a time t at which $m \in PENDING_p$ such that $m.stage_p = s_0$, then all correct processes $q \in group(p)$ eventually execute line 18 such that $m \in msgSet' \wedge m.stage = s_0$.*
- *(b) if there exists a time t at which $m \in PENDING_p$ such that $m.stage_p = s_1$, then for all correct processes $q \in m.dest$, m eventually reaches stage s_1 on q.*
- *(c) if there exists a time t at which $m \in PENDING_p$ such that $m.stage_p = s_2$, then all correct processes $q \in group(p)$ eventually execute line 18 such that $m \in msgSet' \wedge m.stage = s_2$.*
- *(d) if there exists a time t at which $m \in PENDING_p$ such that $m.stage_p = s_1$, then for all correct processes $q \in m.dest$, m eventually reaches stage s_3 on q.*

Proof:

- (a) By Lemma A.6, eventually $m \in PENDING_q$. Suppose, by way of contradiction, that there exists a correct process $r \in group(p)$ that never decides on m at line 18 such that $m.stage = s_0$. Therefore, by by Lemma A.2, and by the uniform agreement and termination properties of consensus, there exists no correct process $q \in group(p)$ that decides on m with $m.stage = s_0$. Consequently, m never reaches stage s_1 on process q and no process q A-Delivers m. Therefore for all q, $m \in PENDING_q \wedge m.stage = s_0$ holds forever. Let t be the time after which all faulty processes have crashed. Therefore, after t, (*) processes q always propose a set of messages $msgSet$ such that $m \in msgSet \wedge m.stage = s_0$ at line 16. By the termination property of consensus, processes q execute an infinite number of consensus instances. Therefore by (*), and by the uniform integrity and uniform agreement properties of consensus, processes q (including r) eventually decide on m such that $m.stage = s_0$, a contradiction.

- (b) Because p sets m's stage to s_1 at line 23, p decided on m in an instance k such that $m.stage = s_0$. There are two cases to consider: (b-i) $q \in group(p)$ and (b-ii) $q \notin group(p)$.
  - In case (b-i), by the uniform agreement property of consensus and by Lemma A.2, q eventually decides on m in instance k such that $m.stage = s_0$. Consequently, q eventually sets m's stage to s_1 at line 23.
  - In case (b-ii), p sends a (TS, m) message to all $q \in m.dest \setminus group(p)$. Because p is correct and links are quasi-reliable, (*) q eventually receive that message.
    We now prove that for all groups in $m.dest \setminus group(p)$ there exists at least one process r such that there exists a time at which r adds m to $PENDING_r$ with $m.stage = s_0$ at line 13. By (a), this shows that m eventually transitions to stage s_1 on all correct processes $q \in m.dest \setminus group(g)$. Suppose, by way of contradiction, that there exists a group $g \in (m.dest \setminus group(p))$ in which no process adds m to $PENDING$ at line 13. Consequently, because consensus instances are executed inside groups, in g, no process adds m to $PENDING$ or $ADELIVERED$. Therefore, by (*) all processes in g eventually execute line 13, a contradiction.

- (c) We first prove that m eventually reaches stage s_2 on q. If $m.stage = s_2$ on p, then m reached stage s_1 on p before. Therefore by (b), m eventually reaches stage s_1 on all correct processes $r \in m.dest$. Therefore, r sends a (TS, m) message to all processes in $m.dest \setminus group(r)$ and because there is at least one correct process per group and links are quasi-reliable, all processes $q \in group(p)$ eventually receive (TS, m) messages from every group different from $group(p)$. As m reaches stage s_2 on p, on every correct process $q \in group(p)$, line 35 evaluates to false, and m reaches stage s_2 on all q.
  Now suppose, by way of contradiction, that there exists a correct process $s \in group(p)$ that never decides on m at line 18 such that $m.stage = s_2$. Consequently, by Lemma A.2, and by the uniform agreement and termination properties of consensus, no process in $group(p)$ decides on m such that $m.stage = s_2$ and none A-Delivers m. Therefore for all correct processes $q \in group(p)$, $m \in PENDING_q \wedge m.stage = s_2$ holds forever. Let t be the time after which all faulty processes have crashed. Therefore, after t, (*) processes q always propose a set of messages $msgSet$ such that $m \in msgSet \wedge m.stage = s_2$ at line 16. By Lemma A.2 and the termination property of consensus, processes q execute an infinite number of consensus instances. Therefore, by (*), and the uniform integrity and uniform agreement properties of consensus, processes q (including s) eventually decide on m such that $m.stage = s_2$, a contradiction.

- (d) If there exists a time at which, on p, $m \in PENDING_p$ such that $m.stage = s_1$, then by (b), on all correct processes $q \in m.dest$ m reaches stage s_1 and q executes line 24. Because there is at least one correct process per group, links are quasi-reliable, and processes p and q are correct, q eventually executes line 33. There are two cases to consider: on q, either (i) line 35 evaluates to true or (ii) not.
  - In case (i), Lemma A.7-(d) trivially holds from the algorithm.
  - In case (ii), m reaches stage s_2 on q. By (c), q eventually decide on m such that $m.stage = s_2$. Therefore, m eventually reaches stage s_3 at line 26. □

Lemma A.8 *For any message m and any correct process p, if there exists a time at which $m \in PENDING_p$, then m eventually reaches stage s_3 on p.*

Proof: There are two cases to consider, either (a) $|m.dest| = 1$, or (b) $|m.dest| > 1$:

- In case (a), p adds m to $PENDING_p$ (a-i) at line 13 or (a-ii) at line 30.

- In case (a-i), by Lemma A.7-(a), all correct processes $q \in group(p)$ (including p) eventually decide on a consensus instance such that $m \in msgSet' \wedge m.stage = s_0$ and on p, $m.stage$ is set to s_3 at line 29.
- In case (a-ii), Lemma A.8 trivially holds from the algorithm.

- In case (b), p adds m to $PENDING_p$ (b-i) at line 13 or (b-ii) at line 30.
  - In case (b-i), by Lemma A.7-(a), all correct processes in $q \in group(p)$ eventually execute line 18 such that $m \in msgSet' \wedge m.stage = s_0$. Because there exists at least one correct process in each group, there is at least one correct process $r \in group(p)$ such that m reaches stage s_1 at line 23. Consequently, by Lemma A.7-(d), m reaches stage s_3 on all correct processes in $m.dest$ (including p).
  - In case (b-ii), when p adds m to $PENDING_p$, either (b-ii-1) $m.stage = s_1$ or (b-ii-2) $m.stage = s_3$.
    - In case (b-ii-1), by Lemma A.7-(d), on all correct processes $q \in m.dest$ (including p), m eventually reaches stage s_3.
    - In case (b-ii-2), Lemma A.8 holds. □

Lemma A.9 *For any correct process p and any message m such that there exists a time at which $m \in PENDING_p$, after m reaches stage s_3 on p, p eventually stops adding messages m' to $PENDING_p$ such that $m'.ts \leq m.ts_p^{s_3}$.*

Proof: Message m reaches stage s_3 at line 26, at line 29, or at line 36. In the three cases, from line 31, there exist a time t at which $K_p > m.ts_p^{s_3}$. After t, p can add a message m' to $PENDING_p$ either (a) at line 13 or (b) at line 30.

- In case (a), before adding m' to $PENDING_p$, m'.ts is set to K_p.
- In case (b), there are three subcases to consider, (c-i) $|m'.dest| > 1 \wedge m'.stage = s_1$, (c-ii) $|m'.dest| > 1 \wedge m'.stage = s_3$, or (c-iii) $|m'.dest| = 1$.
  - In cases (c-i) and (c-iii), $m'.ts$ is set to K_p.
  - In case (c-ii), suppose, by way of contradiction, that p adds an infinite number of messages m' at line 30 such that $|m'.dest| > 1$, $m'.stage = s_3$, and $m'.ts < m.ts_p^{s_3}$. Therefore, by the uniform integrity property of consensus and because $|\Pi| < \infty$, there exists a process $r \in group(p)$ that proposes messages m' such that $m'.stage = s_2$ and $m'.ts < m.ts_p^{s_3}$ an infinite number of times. After such a message m' is decided in consensus, m' transitions to stage s_3. By Lemma A.4, m' can do so at most once and consequently r adds an infinite number of different messages m'' to $PENDING_r$ such that $m''.stage = s_1 \wedge |m''.dest| > 1 \wedge m''.ts < m.ts_p^{s_3}$, a contradiction to (c-i). □

Proposition A.2 (Uniform Agreement) *For any message m, if a process p A-Delivers m, then all correct processes $q \in m.dest$ eventually A-Deliver m.*

Proof: We first show that eventually $m \in PENDING_q$. There are two cases to consider, either (a) $|m.dest| = 1$ or (b) $|m.dest| > 1$. In both cases, because p A-Delivers m, there exists a consensus

instance k such that p decides on m in k with $m.stage = s_0$. By Lemma A.2 and by the uniform agreement of consensus, all correct processes $q \in group(p)$ eventually decide on m in k. Therefore, q adds m to $PENDING_q$ at line 30. This shows the claim for case (a). In case (b), because there is at least one correct process per group, there exists at least one process $r \in group(p)$ that sends (TS, m) to all processes in $(m.dest \setminus group(p))$ at line 24. Therefore, because links are quasi-reliable, all correct processes $q \in (m.dest \setminus group(p))$ eventually receive that message and add m to $PENDING_q$ at line 13 if $m \notin PENDING_q \cup ADELIVERED_q$. Note that if $m \in PENDING_q$, then obviously there is a time at which $m \in PENDING_q$.

By Lemma A.8, m eventually reaches stage s_3 on q. By Lemma A.9, q eventually stops adding messages m' to $PENDING_q$ such that $m'.ts \leq m.ts_q^{s_3}$. By Lemma A.8, all such messages m' eventually reach stage s_3 and are removed from $PENDING_q$. Therefore, q eventually A-Delivers m. □

Proposition A.3 (Validity) *If a correct process p A-MCasts m, then all correct processes $q \in m.dest$ eventually A-Deliver m.*

Proof: We first prove that q eventually adds m to $PENDING_q$. By the properties of Reliable Multicast and because p is correct, all correct processes $q \in m.dest$ R-Deliver m and add m to $PENDING_q$ at line 13 if $m \notin PENDING_q \cup ADELIVERED_q$. Notice that if $m \in ADELIVERED_q$, then obviously q A-Delivered m and Proposition A.3 holds. By Lemma A.8, m eventually reaches stage s_3 on q. By Lemma A.9, q eventually stops adding messages m' to $PENDING_q$ such that $m'.ts \leq m.ts_q^{s_3}$. By Lemma A.8, all such messages m' eventually get to stage s_3 and are removed from $PENDING_q$. Therefore, q eventually A-Delivers m. □

Lemma A.10 *For any message m and any two processes p and q such that p and q A-Deliver m, $m.ts_p^{s_3} = m.ts_q^{s_3}$.*

Proof: There are two cases to consider: either (a) $|m.dest| = 1$ or (b) $|m.dest| > 1$.

- In case (a), by the uniform agreement property of consensus and by Lemma A.4, all processes in p's group decide on m in the same consensus instance k and only in k. Therefore, p and q set $m.ts$ to the same value at line 29.
- In case (b), by the uniform agreement of consensus and by Lemma A.4 for all groups $g \in m.dest$, all processes q in group g decide on m such that $m.stage = s_2$ in the same and only consensus instance k and send the same timestamp at line 24. Therefore, by line 35 and line 39, $m.ts_p^{s_3}$ and $m.ts_q^{s_3}$ are set to the same value. □

Lemma A.11 *For any two messages m_1 and m_2, $m_1 < m_2 \Rightarrow (m_1.ts_p^{s_3}, m_1.id) < (m_2.ts_p^{s_3}, m_2.id)$.*

Proof: Notice that in the proof below, we use the fact that, by definition, for any two messages m_1 and m_2, $m_1.ts < m_2.ts \Rightarrow (m_1.ts, m_1.id) < (m_2.ts, m_2.id)$. Let p be the process that A-Delivers m_1 before m_2. At the time m_1 is A-Delivered, either (a) $m_2 \in PENDING_p$ or (b) $m_2 \notin PENDING_p$.

- In case (a), $(m_1.ts_p^{s_3}, m_1.id) < (m_2.ts_p^{s_3}, m_2.id)$ holds trivially by the condition of line 4.
- In case (b), because m_2 is not in $PENDING_p$ at the time m_1 is A-Delivered, a message is removed from this set only after it has been A-Delivered (line 7), and m_2 is A-Delivered after m_1, (*) m_2 has not yet been added to $PENDING_p$ either at line 13 or at line 30. Since K increases after each consensus instance, if $|m_2.dest| = 1$, $m_1.ts_p^{s_3} < m_2.ts_p^{s_3}$. If $|m_2.dest| > 1$, before m_2 reaches stage s_3 on p, p executed line 18 such that $m_2 \in msgSet' \wedge m_2.stage = s_0$ and m_2 transitions to stage s_1 at line 23. Therefore, since K increases after each consensus instance and because of (*), at the time

m_2 reaches stage s_1, $m_1.ts_p^{s_3} < m_2.ts$. By line 35 or line 39 we have that $m_2.ts_p^{s_3}$ is equal to the maximum of all timestamps received, therefore $m_1.ts_p^{s_3} < m_2.ts_p^{s_3}$.

Lemma A.12 *For any two processes p, q, and any two messages m_1, m_2 such that $p, q \in m_1.dest \cap m_2.dest$, if p A-Delivers m_1 before m_2 and q A-Delivers m_2, then q A-Delivers m_1 before.*

Proof: Because there is at least one correct process per group and by Proposition A.2, there exists a correct process $r \in group(q)$ that A-Delivers m_1 and m_2. Let k_1 and k_2 be the largest K such that $group(q)$ decides on m_1 in consensus instance k_1 and on m_2 in consensus instance k_2 respectively (the decision of instance k_1 (k_2) is such that m_1's (m_2's) stage is either s_0 or s_2). From the algorithm, $k_1 = m_1.ts_r^{s_3}$. By Lemma A.10, $m_1.ts_r^{s_3} = m_1.ts_p^{s_3}$. By Lemma A.11 and because p A-Delivers m_1 before m_2, (*) $(m_1.ts_p^{s_3}, m_1.id) < (m_2.ts_p^{s_3}, m_2.id)$, thus $m_1.ts_p^{s_3} \leq m_2.ts_p^{s_3}$. By Lemma A.10, $m_2.ts_p^{s_3} = m_2.ts_r^{s_3}$. From the algorithm, $k_2 = m_2.ts_r^{s_3}$, therefore, (**) $k_1 \leq k_2$. Hence by Lemma A.1, q decides instance k_1 and thus, q adds m_1 to $PENDING_q$ at line 30 before A-Delivering m_2. There are two cases to consider, either the decision of instance k_1 is such that m_1's stage is (a) s_0 or (b) s_2.

- In case (a), there are two cases to consider, either (a-i) $|m.dest| = 1$ or (a-ii) $|m.dest| > 1$.
  - In case (a-i), on q, m_1 transitions to stage s_3 before q A-Delivers m_2. Because $k_1 = m_1.ts_q^{s_3} \leq k_2 = m_2.ts_q^{s_3}$, by (*) and Lemma A.10, $(m_1.ts_q^{s_3}, m_1.id) < (m_2.ts_q^{s_3}, m_2.id)$. Therefore, from the condition of line 4, q A-Delivers m_1 before m_2.
  - In case (a-ii), on q, m_1 transitions to stage s_1 before q A-Delivers m_2. Since $m_1.ts_q^{s_1} = k_1$ and k_1 is the last consensus instance in $group(q)$ that decides on m_1, on q, after m_1 transitions to stage s_1, m_1's timestamp does not change. By Lemma A.10, $k_1 = m_1.ts_p^{s_3}$ and $m_2.ts_p^{s_3} = m_2.ts_q^{s_3} = k_2$. Hence, since $k_1 \leq k_2$, by (*), q A-Delivers m_1 before m_2, otherwise from the condition of line 4, q would never A-Deliver m_2.
- In case (b), the same argument as in (a-i) is used. □

Lemma A.13 *For any two processes p and q and any time t, either $P_{p,q}(S_p^t) \subseteq P_{p,q}(S_q^t)$ or $P_{p,q}(S_q^t) \subseteq P_{p,q}(S_p^t)$.*

Proof: We prove Lemma A.13 by showing that $\exists m_1 \in P_{p,q}(S_p^t) : m_1 \notin P_{p,q}(S_q^t) \Rightarrow \forall m_2 \in P_{p,q}(S_q^t) : m_2 \in P_{p,q}(S_p^t)$. Suppose, by way of contradiction, that $\exists m_1 \in P_{p,q}(S_p^t) : m_1 \notin P_{p,q}(S_q^t) \wedge \exists m_2 \in P_{p,q}(S_q^t) : m_2 \notin P_{p,q}(S_p^t)$. Because there is at least one correct process per group and by Proposition A.2, there exist correct processes $r \in group(p)$ and $s \in group(q)$ such that (*) r and s A-Deliver m_1 and m_2. There are two cases to consider, either (a) r A-Delivers m_1 before m_2 or (b) the opposite.

- In case (a), because $m_2 \in P_{p,q}(S_q^t)$, by (*), and by Lemma A.12, q A-Delivers m_1 before A-Delivering m_2 and therefore $m_1 \in P_{p,q}(S_q^t)$, a contradiction.
- In case (b), because $m_1 \in P_{p,q}(S_p^t)$, by (*), and by Lemma A.12, p A-Delivers m_2 before A-Delivering m_1 and therefore $m_2 \in P_{p,q}(S_p^t)$, a contradiction. □

Proposition A.4 (Uniform Prefix Order) *For any two processes p and q and any time t, either $P_{p,q}(S_p^t)$ is a prefix of $P_{p,q}(S_q^t)$ or $P_{p,q}(S_q^t)$ is a prefix of $P_{p,q}(S_p^t)$.*

Proof: We proceed by induction on the length l of $P_{p,q}(S_p^t)$.

- Base step ($l = 0$): $P_{p,q}(S_p^t) = \epsilon$ and since ϵ is a prefix of all sequences (including the empty sequence), $P_{p,q}(S_p^t)$ is a prefix of $P_{p,q}(S_q^t)$.

- Induction step: Suppose that Proposition A.12 holds for $l-1$, we prove that Proposition A.12 holds for l. We do so by showing that $\neg(P_{p,q}(S_p^t)$ is a prefix of $P_{p,q}(S_q^t)) \Rightarrow P_{p,q}(S_q^t)$ is a prefix of $P_{p,q}(S_p^t)$. Suppose, by way of contradiction, that (*) $\neg(P_{p,q}(S_p^t)$ is a prefix of $P_{p,q}(S_q^t)) \wedge \neg(P_{p,q}(S_q^t)$ is a prefix of $P_{p,q}(S_p^t))$. By the induction hypothesis, either (a) $\exists\alpha : P_{p,q}(S_{p_{l-1}}^t) \oplus \alpha = P_{p,q}(S_q^t)$ or (b) $\exists\beta : P_{p,q}(S_q^t) \oplus \beta = P_{p,q}(S_{p_{l-1}}^t)$.[9] We now show that (a) and (b) lead to a contradiction.
  - In case (a), $P_{p,q}(S_{p_{l-1}}^t) = \{m_1, .., m_{l-1}\}$, $P_{p,q}(S_p^t) = \{m_1, .., m_l\}$, and $P_{p,q}(S_q^t) = \{m_1, .., m_{l-1}\} \oplus \alpha'$. There are two cases to consider, (a-i) $\alpha' = \epsilon$ or (a-ii) $\alpha' \neq \epsilon$.
    - In case (a-i), $\alpha' = \epsilon$ and thus $P_{p,q}(S_{p_{l-1}}^t) = P_{p,q}(S_q^t)$. Consequently, $P_{p,q}(S_q^t)$ is a prefix of $P_{p,q}(S_p^t)$, a contradiction to (*).
    - In case (a-ii), because $\neg(P_{p,q}(S_p^t)$ is a prefix of $P_{p,q}(S_q^t))$, m_l is not the first message in α', let m_a be that message. Hence, $P_{p,q}(S_p^t) \not\subseteq P_{p,q}(S_q^t)$ and $P_{p,q}(S_q^t) \not\subseteq P_{p,q}(S_p^t)$, a contradiction to Lemma A.13.
  - In case (b), $\exists\beta : P_{p,q}(S_q^t) \oplus \beta = P_{p,q}(S_{p_{l-1}}^t)$ and therefore $P_{p,q}(S_q^t)$ is a prefix of $P_{p,q}(S_p^t)$, a contradiction to (*). □

A.3.2 The Proof of Algorithm $\mathcal{A}2$

Definition A.3 *We define $msgsToADel_p^k$ as the value of set $msgsToADel_p$ after process p executed line 18 when $K_p = k$. If process p does not execute line 18 when $K_p = k$, $msgsToADel_p^k = \perp$.*

Lemma A.14 *For any k, any two processes p and q such that group(p) = group(q) and any two messages $(k, msgSet'_p)$ and $(k, msgSet'_q)$ respectively sent by p and q at line 15, $msgSet'_p = msgSet'_q$.*

Proof: Follows directly from the uniform agreement property of consensus. □

Lemma A.15 *For any two processes p and q and any k, if p and q execute line 18 when $K_p = K_q = k$, then $msgsToADel_p^k = msgsToADel_q^k$.*

Proof: From the condition of line 16, p and q received a message (k, -) from a process in each group different from $group(p)$ and $group(q)$. By Lemma A.14, each two messages $(k, msgSet'_r)$ and $(k, msgSet'_s)$ coming from processes r and s that are in the same group are such that $msgSet'_r = msgSet'_s$. There are two cases to consider, either (a) $group(p) = group(q)$ or (b) not.

- In case (a), by the uniform agreement property of consensus, p and q add the same set of messages to $Msgs$ at line 17. Therefore, $msgsToADel_p^k = msgsToADel_q^k$.
- In case (b), let $msgSet'_p$ and $msgSet'_q$ be the set of messages that p and q respectively add to $Msgs$ at line 17. By the condition of line 16, p received a message (k, $msgSet_1$) from a process in $group(q)$ and q received a message (k, $msgSet_2$) from a process in $group(p)$. Because processes send the same set of messages at line 15 that they add to $Msgs$ at line 17, by Lemma A.14, $msgSet'_p = msgSet_2$ and $msgSet'_q = msgSet_1$. Therefore, $msgsToADel_p^k = msgsToADel_q^k$. □

Proposition A.5 (Uniform Integrity) *For any process p and any message m, (a) p A-Delivers m at most once and (b) only if m was previously A-BCast.*

Proof:

[9] $P_{p,q}(S_{p_{l-1}}^t)$ denotes the prefix of $P_{p,q}(S_p^t)$ of length $l-1$.

- (a) Let k be the value of K_p the first time p A-Delivers m. Consequently, $m \in msgsToADel_p^k$ when $K_p = k$ at line 18 (notice that m can only appear once in $msgsToADel$ because it is a set). By Lemma A.15, all processes q that execute line 18 when $K_q = k$ are such that $m \in msgsToADel_q^k$. Consequently, since processes add m to $ADELIVERED$ at line 20 after A-Delivering m, no process proposes m to a consensus instance $k' > k$. Therefore, there exists no $k' > k$ such that $m \in msgsToADel_p^{k'}$ and p never A-delivers m again.
- (b) Follows directly from the algorithm. □

Lemma A.16 *For any process p and any k,*

- *(a) if p decides in consensus instance k, then all correct processes q eventually decide in instance k.*
- *(b) p does not wait forever at line 16 when $K_p = k$.*

Proof: We proceed by simultaneous induction on (a) and (b).

- Base step ($k = 1$):
  - (a) There are two cases to consider: either (a-1) $q \in group(p)$ or (a-ii) not.
    - (a-1) Variable K is initialized to 1. Therefore by the uniform agreement property of consensus, all correct processes q eventually decide in instance 1.
    - (a-2) By (a-1) and because there is at least one correct process per group, there is at least one correct process $r \in group(p)$ that sends a message (1, -) to all processes $q \notin group(p)$ at line 15. Because r is correct and links are quasi-reliable, all correct processes q eventually receive that message. Thus, eventually, $Barrier_q \geq 1$. Consequently, q proposes a value in instance 1 (if q has not decided yet in instance 1) and by the termination property of consensus, q eventually decides in instance 1.
  - (b) Suppose, by way of contradiction, that p waits forever at line 16. Consequently, p is correct. By (a), all correct processes q eventually decide in instance 1. After deciding in instance 1, correct processes q send a (1, -) message to all processes not in $group(q)$. Because there is at least one correct process per group, p is correct, and links are quasi-reliable, p eventually receive this message from a process in every group different from $group(p)$ and stops waiting at line 16, a contradiction.
- Induction step: Suppose that (a) and (b) hold for $k - 1$ we prove that they hold for k.
  - (a) There are two cases to consider either (a-1) $q \in group(p)$ or (a-ii) not.
    - (a-1) From the induction hypotheses, q eventually decides in instance $k - 1$. Thus, eventually, $K_q = k$. Therefore by the uniform agreement property of consensus, q eventually decide in instance k.
    - (a-2) By (a-1) and because there is at least one correct process per group, there is at least one correct process $r \in group(p)$ that sends a message (k, -) to all processes $q \notin group(p)$ at line 15. Because r is correct and links are quasi-reliable, all correct processes q eventually receive that message. Thus, eventually, $Barrier_q \geq k$. By the induction hypotheses, q decides in instance $k - 1$ and does not wait forever at line 16 when $K_q = k - 1$. Consequently, q proposes a value in instance k (if q has not decided yet in instance k) and by the termination property of consensus, q eventually decides in instance k.
  - (b) The same argument as in the base step of (b) is used, where every occurrence of "1" is replaced by "k". □

Proposition A.6 (Uniform Agreement) *For any message m, if a process p A-Delivers m, then all correct processes q eventually A-Deliver m.*

Proof: If p A-Delivers m, then there exists a k such that $m \in msgsToADel_p^k$. Thus, p decided in consensus instance k. By Lemma A.16, every correct process q eventually decides in consensus instance k and executes line 18 when $K_q = k$. By Lemma A.15, $m \in msgsToADel_q^k$ and therefore q A-Delivers m. □

Lemma A.17 *For any group g and any message m, if there is a time after which all correct processes p in g are such that $m \in RDELIVERED_p$, then all correct processes eventually A-Deliver m.*

Proof: Suppose, by way of contradiction, that there exists a correct process that never A-Delivers m. By Proposition A.6, no correct process A-Delivers m. Therefore, $m \in RDELIVERED_p \setminus ADELIVERED_p$ eventually forever. Consequently, by the termination property of consensus and Lemma A.16, processes p execute an infinite number of consensus instances. Let t be the time at which all faulty processes have crashed. After t, consensus proposals in g always contain m, and thus, by the uniform integrity and uniform agreement properties of consensus, processes p eventually decide on m in an instance k.

Consequently, (*) processes p send a $(k, msgSet'_p)$ message such that $m \in msgSet'_p$. Because there is at least one correct process in each group and links are quasi-reliable, all correct processes $r \in \Pi$ eventually receive that message and eventually $Barrier_r \geq k$. By the termination property of consensus and by Lemma A.16, there exists a time at which processes r have executed line 18 when $K_r = k$. Therefore, by the condition of line 16 and (*), $m \in msgsToADel_r^k$ and thus, all correct processes eventually A-Deliver m, a contradiction. □

Proposition A.7 (Validity) *If a correct process p A-BCasts m, then all correct processes eventually A-Deliver m.*

Proof: By the validity property of reliable multicast, all correct processes q in group(p) eventually R-Deliver m and add m to $RDELIVERED_q$. Therefore, by Lemma A.17, all correct processes eventually A-Deliver m. □

Definition A.4 *We define the round of a message m as the value k such that there exists a process p with $m \in msgsToADel_p^k$. If m is never A-Delivered by any process, $round(m) = \perp$.*[10]

Lemma A.18 *For any process p and any two messages m_1 and m_2 such that $round(m_1) \neq \perp$, $round(m_2) \neq \perp$, and $round(m_1) < round(m_2)$, if p A-Delivers m_2 then p A-Delivers m_1 before.*

Proof: If p A-Delivers m_2, then there exists a time at which $K_p = round(m_2)$. Because K_p is monotonically increasing with time and since $round(m_1) < round(m_2)$, p executes line 19 when $K_p = round(m_1)$ before executing the same line when $K_p = round(m_2)$. Therefore, p A-Delivers m_1 before m_2. □

Proposition A.8 (Uniform Prefix Order) *For any two processes p and q and any time t, either $P_{p,q}(S_p^t)$ is a prefix of $P_{p,q}(S_q^t)$ or $P_{p,q}(S_q^t)$ is a prefix of $P_{p,q}(S_p^t)$.*

Proof: Note that since for any message m, $m.dest = \Gamma$, instead of writing $P_{p,q}(S_p^t)$ and $P_{p,q}(S_q^t)$, we simply write S_p^t and S_q^t respectively. We proceed by induction on the length l of S_p^t.

- Base step ($l = 0$): $S_p^t = \epsilon$ and since ϵ is a prefix of all sequences (including the empty sequence), S_p is a prefix of S_q.

[10] Note that by Proposition A.5 and Lemma A.15, for any message m, round(m) is uniquely defined.

- Induction step: Suppose that Proposition A.8 holds for $x = l-1$, we prove that Proposition A.8 holds for l. We do so by showing that $\neg(S_p^t$ is a prefix of $S_q^t) \Rightarrow S_q^t$ is a prefix of S_p^t. Suppose, by way of contradiction, that (*) $\neg(S_p^t$ is a prefix of $S_q^t) \wedge \neg(S_q^t$ is a prefix of $S_p^t)$. By the induction hypothesis, either (a) $\exists\alpha : S_{p_{l-1}}^t \oplus \alpha = S_q^t$ or (b) $\exists\beta : S_q^t \oplus \beta = S_{p_{l-1}}^t$.[11] We now show that (a) and (b) lead to a contradiction.

  - In case (a), $S_{p_{l-1}}^t = \{m_1, .., m_{l-1}\}$, $S_p^t = \{m_1, .., m_l\}$, and $S_q^t = \{m_1, .., m_{l-1}\} \oplus \alpha'$. There are two cases to consider, (a-i) $\alpha' = \epsilon$ or (a-ii) $\alpha' \neq \epsilon$.
    - In case (a-i), $S_{p_{l-1}}^t = S_q^t$. Thus, S_q^t is a prefix of S_p^t, a contradiction to (*).
    - In case (a-ii), because $\neg(S_p^t$ is a prefix of $S_q^t)$, m_l is not the first message in α', let m_a be that message. Let k and k' be the values of $round(m_l)$ and $round(m_a)$ respectively. There are three cases to consider: (a-ii-1) $k < k'$, (a-ii-2) $k = k'$, or (a-ii-3) $k > k'$.
      - In case (a-ii-1), by Lemma A.18, $m_l \in S_{q_{l-1}}^t$ and thus because $S_{p_{l-1}}^t$ is a prefix of S_q^t, $m_l \in S_{p_{l-1}}^t$. Therefore, p A-Delivers m_l twice, a contradiction to Proposition A.5.
      - In case (a-ii-2), by Lemma A.15, $msgsToADel_p^k = msgsToADel_q^k$. When processes A-Deliver messages of $msgsToADel$ in some deterministic order, either (a-ii-2-*) m_l appears before m_a or (a-ii-2-**) the opposite. In case (a-ii-2-*), using the same argument as (a-ii-1), we conclude that p A-Delivers m_l twice, a contradiction to Proposition A.5. In case (a-ii-2-**), $m_a \in S_{p_{l-1}}^t$ and because $S_{p_{l-1}}^t$ is a prefix of S_q^t, $m_a \in S_{q_{l-1}}^t$. Therefore, q A-Delivers m_a twice, a contradiction to Proposition A.5.
      - In case (a-ii-3), by Lemma A.18, $m_a \in S_{p_{l-1}}^t$ and thus, because $S_{p_{l-1}}^t$ is a prefix of S_q^t, $m_a \in S_{q_{l-1}}^t$. Therefore, q A-Delivers m_a twice, a contradiction to Proposition A.5.
  - In case (b), $\exists\beta : S_q^t \oplus \beta = S_{p_{l-1}}^t$ and therefore S_q^t is a prefix of S_p^t, a contradiction to (*). □

Lemma A.19 *If there exists a time after which no message is A-BCast, then for any correct process p, eventually $(RDELIVERED_p \setminus ADELIVERED_p) = \emptyset$ forever.*

Proof: If there exists a time after which no message is A-BCast, then there exists a time t after which no message is R-MCast. Therefore, by the uniform integrity of reliable multicast, p only R-Delivers a finite number of messages and thus, there exists a time after which no message is added to $RDELIVERED_p$. We now prove that for any message $m \in RDELIVERED_p$, eventually $m \in ADELIVERED_p$. Since $m \in RDELIVERED_p$, p R-Delivered m. By the agreement property of reliable multicast and because p is correct, all correct processes $q \in group(p)$ eventually R-Deliver m. By Lemma A.17, all correct processes eventually A-Deliver m and therefore eventually $m \in ADELIVERED_p$. □

Lemma A.20 *If there exists a time after which no message is A-BCast, then there exists a $Barrier_{max}$ such that for all correct processes p, $Barrier_p < Barrier_{max}$.*

Proof: By Lemma A.19, for all correct processes p, eventually $(RDELIVERED_p \setminus ADELIVERED_p) = \emptyset$ forever. Let t_p be the earliest time at which p executes line 20 such that after executing line 20, $(RDELIVERED_p \setminus ADELIVERED_p) = \emptyset$ forever, and let k_p be the value of K_p at time t_p. We first prove that there exists a k such that for all p, $k_p = k$. Suppose, by way of contradiction, that there exist correct processes q, r such that $k_q > k_r$. Let m be the last message q A-Delivers in instance k_q (such an m exists by the definition of k_q). Before A-Delivering m, q sends a message $(k_q, msgSet'_q)$ such that $m \in msgSet'_q$ to all. Because q and r are correct and links are quasi-reliable, r eventually receives this message and thus eventually $Barrier_r \geq k_q$. By the termination property of consensus and Lemma A.16,

[11] $S_{p_{l-1}}^t$ denotes the prefix of S_p^t of length $l-1$.

r eventually decides in consensus instance k_q. Consequently, r eventually executes line 18 when $K_r = k_q$. By Proposition A.5, r A-Delivers m only once and therefore $k_r \geq k_q$, a contradiction.
Now suppose, by way of contradiction, that there exists a process q such that $Barrier_q$ keeps on increasing. Variable $Barrier_q$ is increased either (a) at line 23 or (b) line 10.

- From the definition of k_p, for every $k > k_p$ such that $msgsToADel_p^k \neq \perp$, $msgsToADel_p^k = \emptyset$. Therefore, q does not increase $Barrier_q$ at line 23 when $K_p > k_p$, a contradiction.
- From (a), there exists no process r such that $Barrier_r > k_p + 1$, and thus q does not receive a (k, -) message at line 8 such that $k > k_p + 1$, a contradiction. □

Proposition A.9 (Quiescence) *If there exists a time after which no message is A-BCast, then eventually all processes stop sending messages.*

Proof: Messages are sent either (a) at line 5 (reliable multicast), (b) at line 12 (consensus), or (c) at line 15 (send). Obviously, only correct processes can send messages forever. Consequently, proving that eventually all correct processes stop executing these lines is enough to show that eventually processes stop sending messages.[12]

- (a) If there exists a time after which no message is A-BCast, eventually no (correct) process executes line 5 anymore.
- (b) From Lemmata A.19 and A.20, for every process p, the condition of line 11 eventually evaluates to false forever. Therefore, p only executes a finite number of times line 12.
- (c) From (b) and the uniform integrity of consensus, p decides in only a finite number of consensus instances and therefore p executes a finite number of times line 15. □

[12]Notice that we here consider consensus and reliable multicast algorithms that are *halting*, i.e., in all runs of the algorithms, there is a time after which all processes stop taking steps and thus only a finite number of messages is sent. Halting algorithms for consensus and reliable multicast can be found in [11] and [6] respectively.