Optimistic Atomic Broadcast: A Pragmatic Viewpoint*

Fernando Pedone⋆ André Schiper†

⋆Hewlett-Packard Laboratories
Software Technology Laboratory
Palo Alto, CA 94304, USA
e-mail: pedone@hpl.hp.com

†Communication Systems Department, EPFL
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
e-mail: andre.schiper@epfl.ch

Abstract

This paper presents the Optimistic Atomic Broadcast algorithm (OPT-ABcast) which exploits the *spontaneous total order property* experienced in local area networks in order to allow fast delivery of messages. The OPT-ABcast algorithm is based on a sequence of stages, and messages can be delivered during a stage or at the end of a stage. During a stage, processes deliver messages *fast*. Whenever the spontaneous total order property does not hold, processes terminate the current stage and start a new one by solving a Consensus problem which may lead to the delivery of some messages. We evaluate the efficiency of the OPT-ABcast algorithms using the notion of delivery latency.

Keywords: optimistic algorithms, atomic broadcast, efficient algorithms, consensus, asynchronous systems

1 Introduction

Atomic Broadcast is a useful abstraction for the development of fault tolerant distributed applications. Understanding the conditions under which Atomic Broadcast is solvable is an important theoretical issue that has been investigated extensively. Solving Atomic Broadcast efficiently is also an important and highly relevant pragmatic issue. We present in this paper the *Optimistic Atomic Broadcast* algorithm (called hereafter OPT-ABcast), which allows processes, in certain cases, to deliver messages *fast*. The idea of our OPT-ABcast algorithm stems from the observation that, with high probability, messages broadcast in a local area network are received

*A preliminary version of this paper appeared in *Proceedings of the 12th International Symposium on Distributed Computing* (*DISC'98*, pp. 318-322).

totally ordered (e.g., when network broadcast or IP-multicast are used). We call this property *spontaneous total order*. Our algorithm exploits this observation: whenever the spontaneous total order property holds, the OPT-ABcast algorithm delivers messages fast.

The OPT-ABcast algorithm is based on the reduction of Atomic Broadcast to Consensus proposed in [4]. However, contrary to [4], in the OPT-ABcast algorithm, Consensus is not always required to deliver messages. Processes executing the OPT-ABcast algorithm see the system evolve as a sequence of stages, and Consensus is only necessary when processes move from one stage to the next. For any stage k, messages can be delivered by some process p, either (1) *during stage* k (i.e., before p executes Consensus), or (2) *at the end of stage* k (i.e., after p terminates the k-th Consensus execution). Messages can be delivered much quickly during a stage than at the end of a stage, since messages delivered during a stage do not require Consensus. We evaluate the efficiency of the OPT-ABcast algorithm using the notion of *delivery latency*. The efficiency of the OPT-ABcast algorithm is directly related to the spontaneous total order property: the event that triggers the termination of a stage is the violation of this property.

The rest of the paper is structured as follows. Section 2 describes related work, and Section 3 is devoted to the system model and to the definition of delivery latency. In Section 4 we present an overview of the results. Section 5 describes the OPT-ABcast algorithm, and Section 6 discusses its efficiency. Failure handling is discussed in Section 7. Section 8 concludes the paper.

2 Related Work

This work is at the intersection of two domains: (1) Atomic Broadcast algorithms, and (2) optimistic algorithms.

The literature on Atomic Broadcast algorithms is abundant (e.g., [1], [3], [4], [5], [7], [10], [13], [16]). However, the multitude of different models (synchronous, asynchronous, etc.) and assumptions needed to prove the correctness of the algorithms renders any fair comparison difficult. We base our solution on the Atomic Broadcast algorithm as presented in [4] because it provides a theoretical framework that permits to develop the correctness proofs under assumptions that are realistic in many real systems (i.e., unreliable failure detectors).

Optimistic algorithms have been widely studied in transaction concurrency control (e.g., [2], [11]). To our knowledge, there has been no attempt, prior to this work, to use optimistic approaches for solving agreement problems. The closest to the idea presented in the paper is [8], where the authors reduce the Atomic Commitment problem to Consensus and, in order to have a fast decision, exploit the following property of the Consensus problem: if every process starts Consensus with the same value v, then the decision is v. This paper presents a more general idea, and does not require all the initial values to be equal. Moreover, we have here the typical trade-off of optimistic algorithms: if the optimistic assumption holds, there is a benefit (in efficiency), but if the optimistic assumption does not hold, there is a loss (in efficiency).

3 System Model and Definitions

3.1 System Model

We consider an asynchronous system composed of n processes $\Pi = \{p_1, \ldots, p_n\}$. A process can only fail by crashing (i.e., we do not consider Byzantine failures). A process that never crashes is *correct*, otherwise it is *faulty*. Processes communicate by message passing, and are connected through FIFO Reliable Channels, defined by the two primitives *send*(m) and *receive*(m). Messages are unique and taken from a set $\mathcal{M}$. FIFO Reliable Channels have the following properties: (i) if process q receives message m from p, then p sent m to q *(no creation)*, (ii) q receives m from p at most once *(no duplication)*, (iii) if p sends m to q, and q is correct, then q eventually receives m *(no loss)*, and (iv) if p sends m to q before sending m' to q, then q does not receive m' before receiving m *(FIFO order)*.

Each process p has access to a local failure detector module $\mathcal{D}_p$ that provides (possibly incorrect) information about the processes that have crashed. A failure detector may make mistakes, that is, it may suspect a process that has not failed or never suspect a process that has failed. Failure detectors have been classified according to *accuracy* and *completeness* properties which characterise the mistakes they can make [4]. In this paper, we require *(strong completeness)*, that is, eventually every process that crashes is permanently suspected by every correct process. Except for the Consensus algorithm, the results presented in the paper are independent of the accuracy property of $\mathcal{D}_p$.

An algorithm A is a collection of n deterministic automata, one per process, and computation proceeds in steps of A. In each step, a process can (1) receive a message that was sent to it, (2) query its failure detector module, (3) modify its state, and (4) send a message to a single process [4]. Informally, a run R of A defines a (possibly infinite) sequence of steps of A.

3.2 Consensus

Consensus is defined by the primitives *propose*(v), and *decide*(v), that satisfy the following properties: (i) every correct process eventually decides some value *(termination)*, (ii) every process decides at most once *(uniform integrity)*, (iii) no two correct processes decide differently *(agreement)*, and (iv) if a process decides v, then v was proposed by some process *(uniform validity)*.

Although Consensus is not solvable in purely asynchronous systems [6], several algorithms are known that solve Consensus in asynchronous systems augmented with failure detectors (e.g., [4], [14]). We do not address this issue in the paper, and assume the existence of an algorithm that solves Consensus.

3.3 Reliable Broadcast and Atomic Broadcast

We assume the existence of a Reliable Broadcast, defined by the primitives *R-broadcast*(m) and *R-deliver*(m). Reliable Broadcast satisfies the following properties [9]: (i) if a correct process R-broadcasts a message m, then it eventually R-delivers m *(validity)*, (ii) if a correct process R-delivers a message m, then all correct processes eventually R-deliver m *(agreement)*, and (iii) for every message m, every process R-delivers m at most once, and only if m was previously R-broadcast by *sender*(m) *(uniform integrity)*.

Atomic Broadcast is defined by *A-broadcast*(m) and *A-deliver*(m). In addition to the properties of Reliable Broadcast, Atomic Broadcast satisfies the *total order* property [4]: (iv) if two correct processes p and q A-deliver two messages m and m', then p A-delivers m before m' if and only if q A-delivers m before m'.

3.4 Delivery Latency

In the following, we introduce the *delivery latency* as a measure of the efficiency of algorithms solving any Broadcast problem (defined by the primitives α-broadcast and α-deliver). The delivery latency is a variation of the Latency Degree introduced in [14], which is based on modified Lamport's clocks [12]:

- initially all clocks are zero,
- a *send* event and a *local* event on a process p do not modify p's local clock,
- let $ts(send(m))$ be the timestamp of the $send(m)$ event, and $ts(m)$ the timestamp carried by message m: $ts(m) \stackrel{\text{def}}{=} ts(send(m)) + 1$,
- the timestamp of $receive(m)$ on a process p is the maximum between $ts(m)$ and p's current clock value.

The delivery latency of a message m α-broadcast in run R of an algorithm A solving a Broadcast problem, denoted $dl^R(m)$, is defined as the difference between (1) the largest timestamp of all α-deliver(m) events (at most one per process) in run R and (2) the timestamp of the α-broadcast(m) event in run R. Let π_m^R be the set of processes that α-deliver message m in run R. The delivery latency of m in R is formally defined as

$$dl^R(m) \stackrel{\text{def}}{=} \max_{p \in \pi_m^R} \left(ts(\alpha\text{-deliver}_p(m)) - ts(\alpha\text{-broadcast}(m))\right),$$

where $ts(\alpha\text{-deliver}_p(m))$ and $ts(\alpha\text{-broadcast}(m))$ denote, respectively, the timestamps of the α-*broadcast*(m) and α-*deliver*(m) events.

For example, consider a broadcast algorithm A_b where a process p, willing to broadcast a message m, sends m to all processes, each process q on receiving m sends an acknowledge message $ACK(m)$ to all processes, and as soon as q receives n $ACK(m)$ messages, q delivers m. Let R be a run of A_b, as shown in Figure 1, where m is the only message broadcast in R. In this case, $dl^R(m) = 2$.

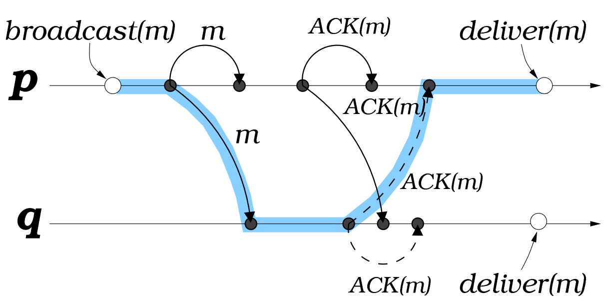

Figure 1: $dl^R(m) = 2$

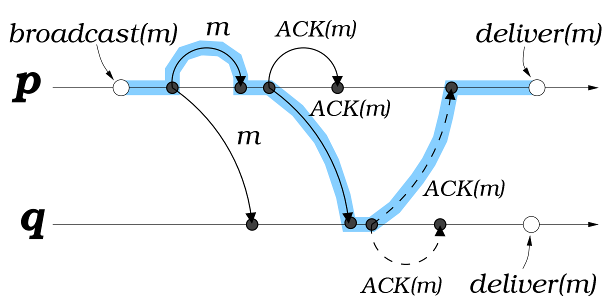

Figure 2: $dl^{R'}(m) = 3$

The delivery latency can be used to measure the α-broadcast(m)-α-deliver(m) "message chain" of a run produced by some Broadcast algorithm A.[1] For example, algorithm A_b requires

[1] A message chain is a sequence $m_1, m_2, ..., m_k$ of messages, such that for each i, $0 \le i \le k$, the receipt of m_i causally precedes the sending of m_{i+1} [15].

that processes send an $ACK(m)$ message only after receiving message m, and so, no run generated by A_b where m is broadcast will have $send_p(ACK(m))$ preceding $receive_p(m)$, for all process p. Nevertheless, algorithm A_b allows a process q to send $ACK(m)$ after having received $ACK(m)$ from some process p (see Figure 2). Thus, there exists a run R' of A_b where m is the only message broadcast, and $receive_q(ACK(m))$ precedes $send_q(ACK(m))$. This leads to $dl^{R'}(m) = 3$.

When characterising a Broadcast algorithm A with the delivery latency parameter, we consider only the set of runs $\mathcal{R}$ produced by A that exhibit the shortest "message chain" (i.e., the smallest delivery latency).

4 Overview of the Results

4.1 OPT-ABcast Algorithm

The OPT-ABcast algorithm exploits the spontaneous total order property: if a process p sends a message m to all processes, and a process q sends a message m' to all processes, then the two messages might be received in the same order by all processes. This property typically holds with high probability in local area networks under normal execution conditions (e.g., moderate load). However, under abnormal execution conditions (e.g., high network loads), this property might be violated. More generally, one can consider that the system passes through periods when the spontaneous total order message reception property holds, and periods when the property does not hold.

Figure 3 illustrates the spontaneous total order property in a system composed of eight workstations (UltraSparc 1+) connected by an Ethernet network (10 Mbits/s). In the experiments, each workstation broadcasts messages to all the other workstations, and receives messages from all workstations over a certain period of time (around 10 sec.). Broadcasts are implemented with IP-multicast, and messages have 1024 bytes. From Figure 3, it can be seen that there is a relation between the time between successive broadcast calls, and the percentage of messages that are received in the same order.

In the OPT-ABcast algorithm, processes progress in a sequence of stages. Messages can be delivered during a stage or at the end of a stage, and the key aspect is that during a stage, messages can be delivered faster than at the end of a stage. In order for a process p to deliver messages during a stage k, p has to determine whether the spontaneous total order property holds. Process p determines whether this property holds by exchanging information about the order in which messages are received (see Figure 4).[2] Once p receives this order information from all the other processes, p uses a *prefix* function to determine whether there is a non-empty common sequence of messages received by all processes.

Whenever the spontaneous total order property does not hold, processes terminate the current stage, and start a new one (see Figure 5). The termination of a stage involves the execution of a Consensus, which can lead to the delivery of messages. Process failures are discussed in Section 7.

[2] In Figures 4 and 5, $\langle m_1, m_2, ...\rangle$ denotes sequence $m_1, m_2, ...$ of messages.

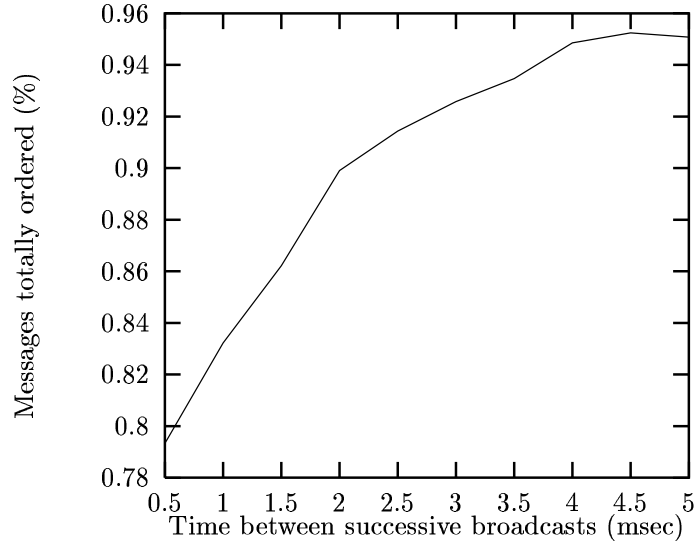

Figure 3: Spontaneous total order property

4.2 Delivery Latency of the OPT-ABcast Algorithm

The notion of efficiency is captured by the delivery latency parameter defined in Section 3.4, which measures the length of the message chain of the OPT-ABcast algorithm between an A-broadcast and an A-deliver. We show that messages delivered during a stage have a delivery latency equal to 2, and messages delivered at the end of a stage have a delivery latency equal to 4. The additional cost *payed* by messages delivered at the end of a stage comes from the Consensus execution. The OPT-ABcast algorithm is based on a Reliable Broadcast and a Consensus, and thus, in order to determine the delivery latency of messages, we use the Reliable Broadcast implementation presented in [4], and the Consensus implementation presented in [14].

Known Atomic Broadcast implementations for the asynchronous model augmented with failure detectors deliver messages with a delivery latency equal to 3. This means that if the spontaneous total order property is violated too frequently, the OPT-ABcast algorithm may become inefficient. However, in case the spontaneous total order property holds frequently, messages can be delivered efficiently using the OPT-ABcast algorithm.

5 The Optimistic Atomic Broadcast Algorithm

5.1 Additional Notation

The OPT-ABcast algorithm presented in the next section handles sequences of messages. In the following we define some terminology needed for the presentation of the algorithm.

A sequence of messages is denoted by $seq = \langle m_1, m_2, \dots \rangle$. We define the operators $\oplus$ and $\ominus$ for concatenation and decomposition of sequences. Let seq_i and seq_j be two sequences of messages. Then, $seq_i \oplus seq_j$ is the sequence of all the messages in seq_i followed by the sequence of all the messages in seq_j, and $seq_i \ominus seq_j$ is the sequence of all the messages in seq_i that are not in seq_j. So, the sequence $seq_i \ominus seq_j$ does not contain any message in seq_j. The prefix function $\odot$ applied to a set of sequences returns the longest common sequence that is a prefix of all the

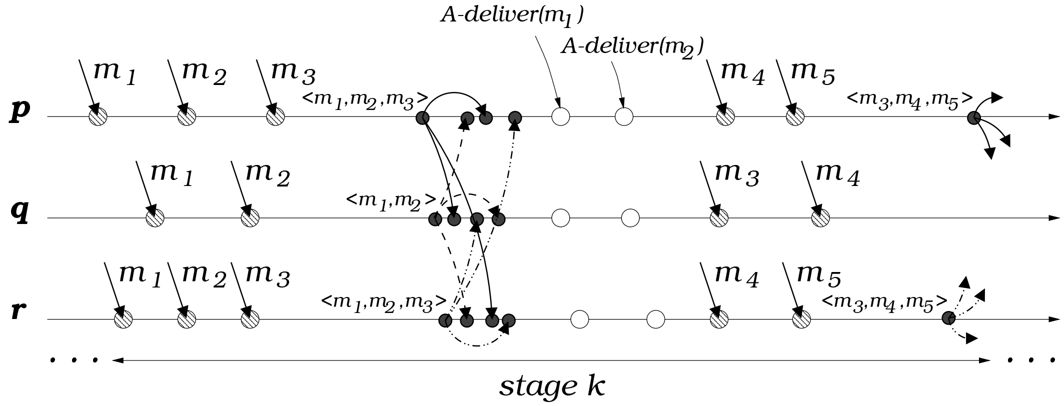

Figure 4: Overview of the OPT-ABcast algorithm (stage k)

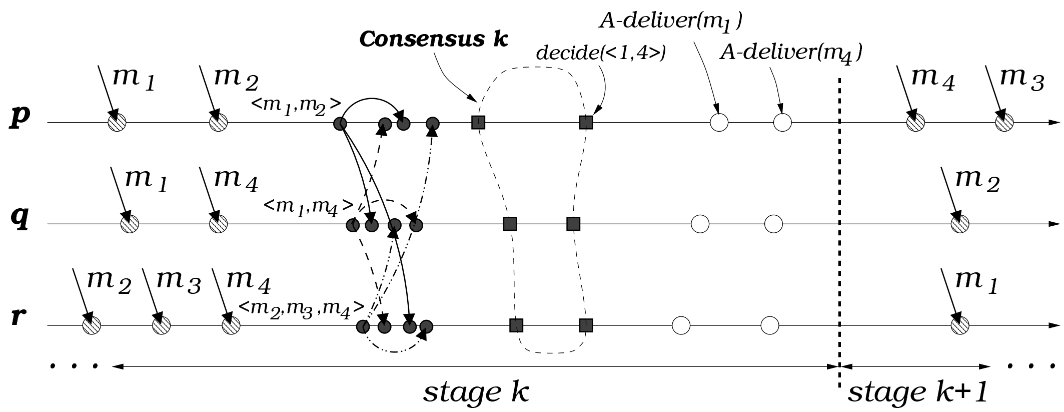

Figure 5: Overview of the OPT-ABcast algorithm (stages k and $k+1$)

sequences, or the empty sequence denoted by ϵ.

For example, if $seq_i = \langle m_1, m_2, m_3\rangle$ and $seq_j = \langle m_1, m_2, m_4\rangle$, then $seq_i \oplus seq_j = \langle m_1, m_2, m_3, m_1, m_2, m_4\rangle$, $seq_i \ominus seq_j = \langle m_3\rangle$, and $\odot(seq_i, seq_j) = \langle m_1, m_2\rangle$.

5.2 Overview of the OPT-ABcast Algorithm

Algorithm 1 (see page 10) solves Atomic Broadcast. Processes executing Algorithm 1 progress in a sequence of local stages numbered $1, ..., k, ...$ Messages A-delivered by a process *during stage* k are included in the sequence $stgA\_deliver^k$. These messages are A-delivered without the cost of Consensus. Messages A-delivered by a process *at the end of stage* k are included in the sequence $endA\_deliver^k$. These messages are A-delivered with the cost of a Consensus execution. We say that a message m is A-delivered *in stage* k if m is A-delivered either during stage k or at the end of stage k.

Every stage k is terminated by a Consensus to decide on a sequence of messages, denoted by $msgStg^k$. Algorithm 1 guarantees that if a correct process starts Consensus (by invoking the *propose* primitive), all correct processes also start Consensus. Notice that if not all correct processes invoke the *propose* primitive in the k-th Consensus execution, then Consensus termination cannot be ensured.

The sequence $msgStg^k$ contains all messages that are A-delivered by every process that reaches the end of stage k. Process p starts stage $k+1$ once it has A-delivered all messages in $endA\_deliver^k$. where $endA\_deliver^k = msgStg^k \ominus stgA\_deliver^k$.

The correctness of Algorithm 1 is based on two properties:

1. for any correct processes p and q, all the messages A-delivered by p in stage k are also A-delivered by q in stage k (i.e., $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$), and

2. every sequence of messages A-delivered by some process p in stage k before p executes Consensus k is a non-empty prefix of the sequence decided in Consensus k (i.e., $stgA\_deliver_p^k$ is a prefix of $msgStg^k$).

5.3 Detailed OPT-ABcast Algorithm

All tasks in Algorithm 1 execute concurrently. At each process p, tasks *GatherMsgs* (lines 11-12) and *TerminateStage* (lines 25-35) are started at initialisation time. Task *StgDeliver*^{k} (lines 13-24) is started by p when p begins stage k. Process p periodically evaluates the condition in line 13, and executes task *StgDeliver*^{k} whenever the sequence $(R\_delivered \ominus A\_delivered) \ominus stgA\_deliver^k$ contain at least one message. Lines 20 and 21 in task *StgDeliver*^{k} are atomic, that is, task *StgDeliver*^{k} is not interrupted (by task *TerminateStage*) after it has executed line 20 and before having executed line 21.

Algorithm 1 uses an "underline" notation (e.g., $\underline{k}$) to specify the type of message a process is waiting for. For example, a process that waits for message $(\underline{k}, msgSeq_q)$ (line 15) will receive a message $(i, -)$ such that $i = k$.

Process p in stage k manages the following sequences.

- $R\_delivered_p$: contains all messages R-delivered by p up to the current time,
- $A\_delivered_p$: contains all messages A-delivered by p up to the current time,
- $stgA\_deliver_p^k$: is the sequence of messages A-delivered by p during stage k, up to the current time,
- $endA\_deliver_p^k$: is the sequence of messages A-delivered by p at the end of stage k.

When p wants to A-broadcast message m, p executes *R-broadcast*(m) (line 9). After p R-delivers a message m (line 11), p includes m in $R\_delivered_p$, and eventually executes task *StgDeliver*^{k} (line 13). The R-deliver at line 11 only R-delivers messages that have been R-broadcast at line 9. At task *StgDeliver*^{k}, p sends a sequence of messages that it has not A-delivered yet to all processes (line 14), and waits for such sequence from all processes (line 15). The next actions executed by p depend on the messages it receives at the *wait* statement (line 15).

1. If p receives a sequence from all processes, and there is a non-empty prefix common to all these sequences, p A-delivers the messages in the common prefix (line 20). If not, p R-broadcasts message (k, EndStg) to terminate the current stage k (line 23).

2. Once p R-delivers message (k, ENDSTG) at line 25, p terminates task $StgDeliver^k$ (line 26), and launches the k-th Consensus execution (line 27), proposing a sequence of all messages p has R-delivered up to the current time but not A-delivered in any stage k', $k' < k$.

3. Upon deciding for Consensus k (line 28), p builds the sequence $endA\_deliver^k$ (line 29) and A-delivers the messages in $endA\_deliver^k$ (line 30). Process p then starts stage $k+1$ (lines 32-35).

5.4 Proof of Correctness

The correctness of the OPT-ABcast algorithm follows from Propositions 5.1 (Agreement), 5.2 (Total Order), 5.3 (Validity), and 5.4 (Integrity). In order to prove some results that follow, we consider the number of times that processes execute lines 13-21 in a given stage. Hereafter, $stgA\_deliver_p^{k,l_k}$ denotes the value of $stgA\_deliver_p^k$ after p executes line 21 for the l_k-th time in stage k, $l_k > 0$, and $stgA\_deliver_p^{k,0}$ denotes the value of $stgA\_deliver_p^k$ before p executes lines 13-21 for the first time (i.e., $stgA\_deliver_p^{k,0} = \epsilon$). Similarly, $prefix_p^l$ and $msgSeq_p^l$ denote, respectively, the values of $prefix_p$ and $msgSeq_p$ after process p executes lines 17 and 15 for the l-th time in a given stage.

Lemma 5.1 *If p and q are two processes that execute the l_k-th iteration of line 21 in stage k, then $stgA\_deliver_p^{k,l_k} = stgA\_deliver_q^{k,l_k}$.*

PROOF. We first show that for any l, $0 < l \leq l_k$, $prefix_p^l = prefix_q^l$. Since p and q execute line 21 for the l-th time in stage k, p and q receive a message of the type $(k, msgSeq)$ from every process in the l-th iteration of lines 15. From line 17 and the fact that communication between processes follows a FIFO order, $prefix_p^l = \odot_{\forall r} msgSeq_r^l$, and $prefix_q^l = \odot_{\forall r} msgSeq_r^l$, where $msgSeq_r^l$ is the l-th message of the type $(k, msgSeq_r)$ received from process r, and we conclude that $prefix_p^l = prefix_q^l$. From line 21, $stgA\_deliver^{k,l} = stgA\_deliver^{k,l-1} \oplus prefix^l$, and a simple induction on l_k leads to $stgA\_deliver_p^{k,l_k} = stgA\_deliver_q^{k,l_k}$. □

Lemma 5.2 *If some process p executes line 21 l times, then all processes in Π execute the send statement at line 14 l times.*

PROOF. This follows directly from the algorithm since p can only execute line 21 after receiving message $(k, msgSeq)$ (line 15) from all processes. Thus, if p executes line 21 l times, it receives message $(k, msgSeq)$ from all processes l times, and from the no creation property of Reliable Channels, all processes execute the $send(k, -)$ statement at line 14 l times. □

Lemma 5.3 *For any process p, and all $k \geq 1$, if p executes the statement $decide(k, msgStg^k)$, then (a) $stgA\_deliver_p^k$ is a prefix of $msgStg^k$, and (b) $stgA\_deliver_p^k$ does not contain the same message more than once.*

PROOF. Assume that p executes $decide(k, msgStg^k)$. By uniform validity of Consensus, there is a process q that executed $propose(k, R\_delivered_q \ominus A\_delivered_q)$, such that $R\_delivered_q \ominus A\_delivered_q = msgStg^k$. Let l_k be the number of times that p executes line 21 before executing $decide(k, -)$. From Lemma 5.2, all processes in Π execute the $send$ statement at line 14 l_k times.

Algorithm 1 OPT-ABcast algorithm

```
Initialisation (see Section 5.3 for a description of the variables):
  R_delivered ← ε
  A_delivered ← ε
  k ← 1
  stgA_deliver^k ← ε
  endA_deliver^k ← ε
  fork tasks { GatherMsgs, StgDeliver^1, TerminateStage }

To execute A-broadcast(m):

  R-broadcast(m)

A-deliver(−) occurs as follows:

  when R-deliver(m)                                              {Task GatherMsgs}
    R_delivered ← R_delivered ⊕ ⟨m⟩

  when (R_delivered ⊖ A_delivered) ⊖ stgA_deliver^k ≠ ε          {Task StgDeliver^k}
    send (k, (R_delivered ⊖ A_delivered) ⊖ stgA_deliver^k) to all
    wait until for [∀q ∈ Π : received (k̲, msgSeq_q) from q or D_p ≠ ∅]
    π = { q | p received (k, msgSeq_q) from q }
    prefix ← ⊙_{∀q∈π} msgSeq_q
    if π = Π and prefix ≠ ε then
      stgDeliver ← prefix ⊖ stgA_deliver^k
      [ deliver all messages in stgDeliver following their order in stgDeliver;
      stgA_deliver^k ← stgA_deliver^k ⊕ prefix ]
    else
      R-broadcast(k, ENDSTG)
      end task

  when R-deliver(k̲, E̲N̲D̲S̲T̲G̲)                                       {Task TerminateStage}
    terminate task StgDeliver^k, if executing
    propose(k, R_delivered ⊖ A_delivered)
    wait until decide(k̲, msgStg^k)
    endA_deliver^k ← msgStg^k ⊖ stgA_deliver^k
    deliver all messages in endA_deliver^k following their order in endA_deliver^k
    A_delivered ← A_delivered ⊕ (stgA_deliver^k ⊕ endA_deliver^k)
    k ← k + 1
    stgA_deliver^k ← ε
    endA_deliver^k ← ε
    fork task StgDeliver^k
```

We show by induction on l_k that $stgA\_deliver_p^{k,l_k}$ is a prefix of $R\_delivered_q \ominus A\_delivered_q$, and $stgA\_deliver_p^{k,l_k}$ does not contain the same message more than once. BASE STEP. ($l_k = 0$) In this case, $stgA\_deliver_p^{k,0} = \epsilon$ and the lemma is trivially true. INDUCTIVE STEP. Assume that the lemma holds for all l'_k, $0 < l'_k < l_k$. We show that $stgA\_deliver_p^{k,l_k}$ is a prefix of $R\_delivered_q \ominus A\_delivered_q$, and $stgA\_deliver_p^{k,l_k}$ does not contain the same message more than once. By line 21, $stgA\_deliver_p^{k,l_k} = stgA\_deliver_p^{k,(l_k-1)} \oplus prefix_p^{l_k}$. Since communication channels are FIFO, any message sent by some process r in the l_k-th execution of *send*$(k, msgSeq_r^{l_k})$ (line 14) is received by p in the l_k-th execution of *receive*$(k, msgSeq_r^{l_k})$ (line 15), and so, after p executes line 17, $prefix_p^{l_k} = \odot_{\forall r} msgSeq_r^{l_k}$. From lines 14 and 15, $msgSeq_r^{l_k} = (R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_r^{k,(l_k-1)}$, and so, $prefix_p^{l_k} = \odot_{\forall r}((R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_r^{k,(l_k-1)})$. By Lemma 5.1, we have $prefix_p^{l_k} = \odot_{\forall r}((R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_p^{k,(l_k-1)})$. Therefore, $stgA\_deliver_p^{k,l_k} = stgA\_deliver_p^{k,(l_k-1)} \oplus (\odot_{\forall r}(R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_p^{k,(l_k-1)})$. From the induction hypothesis, item (a), we have that $stgA\_deliver_p^{k,(l_k-1)}$ is a prefix of $R\_delivered_q \ominus A\_delivered_q$. Furthermore, from item (b) of the induction hypothesis, all messages in $stgA\_deliver_p^{k,(l_k-1)}$ are unique. Thus, $stgA\_deliver_p^{k,l_k} = \odot_{\forall r}(R\_delivered_r \ominus A\_delivered_r)$,[3] and so, $stgA\_deliver_p^{k,l_k}$ is a prefix of $R\_delivered_q \ominus A\_delivered_q$. It also follows that $stgA\_deliver_p^{k,l_k}$ does not contain the same message more than once. For a contradiction, assume that message m is more than once in $stgA\_deliver_p^{k,l_k}$. Thus, for every process r, m is more than once in $R\_delivered_r$. From the algorithm, lines 11 and 12, m has been R-delivered more than once by r, contradicting uniform integrity of Reliable Broadcast. □

Lemma 5.4 *For any two correct processes p and q, and all $k \geq 1$, if p executes line 30 in stage k, then q executes line 30 in stage k.*

PROOF. If p executes line 30 in stage k, then p executes the *decide*$(k, msgStg^k)$ statement at line 28, and the *propose*$(k, -)$ statement at line 27. Therefore, p R-delivers a message of the type (k, ENDSTG) at line 25. By the agreement property of Reliable Broadcast, q eventually R-delivers message (k, ENDSTG), and executes the *propose*$(k, -)$ statement at line 27. By agreement of Consensus, q executes the *decide*$(k, msgStg^k)$ statement, and line 30. □

Lemma 5.5 *For any two processes p and q, and all $k \geq 1$, if both p and q execute line 29, then $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$.*

PROOF. From line 29, $endA\_deliver_p^k = msgStg^k \ominus stgA\_deliver_p^k$, and so, $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_p^k \oplus (msgStg^k \ominus stgA\_deliver_p^k)$. By Lemma 5.3, $stgA\_deliver_p^k$ is a prefix of $msgStg^k$, and so, $stgA\_deliver_p^k \oplus endA\_deliver_p^k = msgStg^k$. From a similar argument, $stgA\_deliver_q^k \oplus endA\_deliver_q^k = msgStg^k$. Therefore, we conclude that $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$. □

Lemma 5.6 *For any process p, and all $k \geq 1$, if message $m \in stgA\_deliver_p^k \oplus endA\_deliver_p^k$ then there is no k', $k' < k$, such that $m \in stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'}$.*

[3] Let seq_i and seq_j be two sequences such that seq_i is a prefix of seq_j, and messages in seq_j are unique. We can show that $seq_i \oplus (seq_j \ominus seq_i) = seq_j$.

PROOF. The proof is by contradiction. Assume that there exist a process p, a message m, some k, and some $k' < k$, such that $m \in stgA\_deliver_p^k \oplus endA\_deliver_p^k$, and $m \in stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'}$. We distinguish two cases: (a) $m \in stgA\_deliver_p^k$, or (b) $m \in endA\_deliver_p^k$. Note that from line 29, it cannot be that $m \in stgA\_deliver_p^k$ and $m \in endA\_deliver_p^k$.

Case (a). From lines 21, 17 and 15 $stgA\_deliver_p^k$ is a common non-empty prefix among the messages of the type $(k, msgSeq)$ received by p from all processes. Thus p has received the message $(k, msgSeq_p)$ (i.e., a message that p sent to itself), such that $m \in msgSeq_p$. But $msgSeq_p = R\_delivered_p \ominus A\_delivered_p$ (line 14), and so, $m \notin A\_delivered_p$. When p executes line 14 at stage k, $A\_delivered_p = \oplus_{i=1}^{k-1}(stgA\_deliver_p^i \oplus endA\_deliver_p^i)$. This follows from line 31, the only line where $A\_delivered$ is updated. Therefore, $m \notin \oplus_{i=1}^{k-1}(stgA\_deliver_p^i \oplus endA\_deliver_p^i)$, contradicting the fact that there is a $k' < k$ such that $m \in stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'}$.

Case (b). From line 29, $m \in msgStg^k$, and from line 28, and validity of Consensus, there is a process q that executes $propose(k, R\_delivered_q \ominus A\_delivered_q)$ such that $m \in R\_delivered_q \ominus A\_delivered_q$. So, $m \notin A\_delivered_q$. Since when q executes line 27, $A\_delivered_q = \oplus_{i=1}^{k-1}(stgA\_deliver_q^i \oplus endA\_deliver_q^i), m \notin \oplus_{i=1}^{k-1}(stgA\_deliver_q^i \oplus endA\_deliver_q^i)$, and from Lemma 5.5 $\oplus_{i=1}^{k-1}(stgA\_deliver_p^i \oplus endA\_deliver_p^i) = \oplus_{i=1}^{k-1}(stgA\_deliver_q^i \oplus endA\_deliver_q^i)$. Thus, $m \notin \oplus_{i=1}^{k-1}(stgA\_deliver_p^i \oplus endA\_deliver_p^i)$, a contradiction that concludes the proof. □

Proposition 5.1 (AGREEMENT). *If a correct process p A-delivers a message m, then every correct process q eventually A-delivers m.*

PROOF: Consider that p has A-delivered message m in stage k. We show that q also A-delivers m in stage k. There are two cases to consider: (a) p A-delivers messages in $endA\_deliver_p^k$, and (b) p does not A-deliver messages in $endA\_deliver_p^k$.

Case (a). From Lemma 5.4 and the fact that p A-delivers messages in $endA\_deliver_p^k$, q A-delivers messages in $endA\_deliver_q^k$, and from Lemma 5.5, $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$. Since p A-delivers m in stage k, $m \in stgA\_deliver_p^k \oplus endA\_deliver_p^k$, and so, $m \in stgA\_deliver_q^k \oplus endA\_deliver_q^k$. Therefore, q either A-delivers m at line 20 (in which case $m \in stgA\_deliver_q^k$), or at line 30 (in which case $m \in stgA\_deliver_q^k$).

Case (b). Since p does not A-deliver messages in $endA\_deliver_p^k$, from Lemma 5.4, no correct process q A-delivers messages in $endA\_deliver_q^k$. However, m is A-delivered in stage k by p, and so, it must be that $m \in stgA\_deliver_p^k$. Assume that $m \in stgA\_deliver_p^{k,l_k}$, where l_k is such that for any $l'_k < l_k$, $m \notin stgA\_deliver_p^{k,l'_k}$. Therefore, p executes the l_k-th iteration of line 21 in stage k, and we claim that q also executes the l_k-th iteration of line 21 in stage k. The claim is proved by contradiction. From the algorithm, q executes $R\text{-}broadcast(k, -)$. By agreement and validity of Reliable Broadcast, every correct process R-delivers the message (k, ENDSTG) and executes $propose(k, -)$. By agreement and termination of Consensus, every correct process decides on Consensus k, and eventually A-delivers messages in $endA\_deliver^k$, contradicting the fact that no correct process A-delivers messages in $endA\_deliver^k$, and concluding the proof of the claim. Since p and q execute the l_k-th iteration of line 21 in stage k, and $m \in stgA\_deliver_p^{k,l'_k}$, from Lemma 5.1, $m \in stgA\_deliver_q^{k,l'_k}$, and from lines 20-21, q A-delivers m. □

Proposition 5.2 (Total Order). *If correct processes p and q both A-deliver messages m and m', then p A-delivers m before m' if and only if q A-delivers m before m'.*

Proof: Assume that p A-delivers message m in stage k, and m' in stage k', $k' > k$. Therefore, $m \in stgA\_deliver_p^k \oplus endA\_deliver_p^k$, and $m' \in stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'}$, and it follows immediately from Lemma 5.5 that q A-delivers m before m'. Now, assume that m and m' are A-delivered by p in stage k. Thus, m precedes m' in $stgA\_deliver_p^k \oplus endA\_deliver_p^k$, and by Lemma 5.5, $stgA\_deliver_p^k \oplus endA\_deliver_p^k = stgA\_deliver_q^k \oplus endA\_deliver_q^k$.

We claim that if m precedes m' in $stgA\_deliver_q^k \oplus endA\_deliver_q^k$, then q A-delivers m before m'. If $m, m' \in stgA\_deliver_q^k$ (respectively $m, m' \in endA\_deliver_q^k$), then, from task *stgDeliver*^{k}, line 20 (respectively *TerminateStage*, line 30), q A-delivers m before m'. Thus, consider that $m \in stgA\_deliver_q^k$ and $m' \in endA\_deliver_q^k$. To reach a contradiction, assume that q A-delivers m' before m. Before A-delivering m at line 20, q executes line 26 and terminates task $stgA\_deliver_q^k$, and so, m cannot be A-delivered in stage k, contradicting that m and m' are A-delivered in stage k, and concluding the proof of the lemma. □

Lemma 5.7 *If a correct process p executes line 25 in stage k, then every correct process q executes line 25 in stage k.*

Proof. The proof is by induction on k. Base step. ($k = 1$) Initially, all correct processes are in stage 1. Thus, if p executes line 25 in stage 1 and R-delivers message $(1, \text{EndStg})$, by the agreement property of Reliable Channels, every correct process eventually executes line 25 and R-delivers message $(1, \text{EndStg})$. Inductive step. Assume that if a correct process p executes line 25 in stage $k-1$, then every correct process q executes line 25 in stage $k-1$. We show that if p executes line 25 in stage k, then q also executes line 25 in stage k. From the algorithm and the termination property of Consensus, after R-delivering message $(k-1, \text{EndStg})$, all correct processes eventually terminate Consensus in stage $k-1$ and execute lines 32-35, starting stage k. Since p R-delivers message (k, EndStg), by agreement of Reliable Channels, every correct process q R-delivers message (k, EndStg). □

Lemma 5.8 *No correct process p has a task $stgDeliver_p^k$, $k > 0$, that is permanently blocked in the wait statement of line 15.*

Proof. For a contradiction, consider that there exists a correct process p such that for some $l_k > 0$, task $stgDeliver_p^k$ is permanently blocked at the l_k-th iteration of line 15. Therefore, (a) there is a process q such that p never receives the message $(k, msgSeq)$ for the l_k-th time from q and (b) $q \notin \mathcal{D}_p$. From (b), and the completeness property of $\mathcal{D}_p$, q is a correct process. From Lemma 5.7, if p executes line 25 in stage k, then q executes line 25 in stage k, but since p never receives $(k, msgSeq)$ for the l_k-th time from q, by the no loss property of Reliable Channels, q does not send message $(k, msgSeq)$ for the l_k-th time to p (line 14).

We now prove the following claim: if q does not execute *send*$(k, msgSeq)$ for the l_k-th time, q executes *R-deliver*(k, EndStg). When p executes the wait statement for the l_k-th time in stage k, there exists a message m such that $m \in (R\_delivered_p \ominus A\_delivered_p) \ominus stgA\_deliver_p^{k,(l_k-1)}$. So, (1a) $m \notin stgA\_deliver_p^{k,(l_k-1)}$, and from line 31, (1b) $m \notin \oplus_{i=1}^{k-1} stgA\_deliver_p^i \oplus endA\_deliver_p^i$. If q does not send the message $(k, msgSeq)$ for the l_k-th time to p, then either (i) $(R\_delivered_q \ominus (\oplus_{i=1}^{k-1}(stgA\_deliver_q^i \oplus endA\_deliver_q^i))) \ominus stgA\_deliver_q^{k,(l_k-1)}$ is empty (line 13) or (ii) task

$stgDeliver_q^k$ is terminated before q sends message $(k, msgSeq)$ for the l_k-th time to p (i.e., q terminates stage k). Furthermore, since p executes line 15 for the l_k-th time, p has executed the $(l_k - 1)$-th iteration of lines 13-21, and received a message from all processes at line 15 for the $(l_k - 1)$-th time. Thus, every process executes the send statement at line 14 at least $l_k - 1$ times, and, from Lemma 5.1, (2a) $stgA\_deliver_p^{k,(l_k-1)} = stgA\_deliver_q^{k,(l_k-1)}$. From Lemma 5.5, (2b) for all $k', 1 \leq k' < k$, $stgA\_deliver_p^{k'} \oplus endA\_deliver_p^{k'} = stgA\_deliver_q^{k'} \oplus endA\_deliver_q^{k'}$. From (1a) and (2a), we conclude that $m \notin stgA\_deliver_q^{k,(l_k-1)}$, and, from (1b) and (2b), $m \notin \oplus_{i=1}^{k-1} stgA\_deliver_q^i \oplus endA\_deliver_q^i$. Since q does not send message $(k, msgSeq)$ for the l_k-th time to p at line 14, m will never be in $R\_delivered_q$. However, by the agreement property of Reliable Broadcast, eventually $m \in R\_delivered_q$ (item (i) of the claim is false), and so, task $stgDeliver_q^k$ is terminated at line 24 or 26 before q sends message $(k, msgSeq)$ for the l_k-th time to p (item (ii) of the claim is true), and q executes $R\text{-}deliver(k, \text{EndStg})$, concluding our claim.

By the agreement of Reliable Broadcast, p eventually R-delivers message (k, EndStg), and so, p executes line 26 and terminates task $stgDeliver_p^k$, contradicting our initial hypothesis that task $stgDeliver_p^k$ remains permanently blocked. □

Proposition 5.3 (Validity). *If a correct process p A-broadcasts a message m, then p eventually A-delivers m.*

Proof: For a contradiction, assume that p A-broadcasts m but never A-delivers it. From Proposition 5.1, no correct process A-delivers m. Since p A-broadcasts m, it R-broadcasts m, and from the validity of Reliable Broadcast, p eventually R-delivers m and includes m in $R\_delivered_p$. Since no correct process A-delivers m, $m \notin A\_delivered_p$, and for all k, $m \notin stgA\_deliver^k$, $k > 0$. From the agreement of Reliable Broadcast, there is a stage k_1 such that for all $l \geq k_1$, and every correct process q, $m \in (R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_q^l$.

Let k_2 be a stage such that for all $l \geq k_2$ every faulty process has crashed (i.e., no faulty process executes stage l), and let $k \geq max(k_1, k_2)$. Thus, no faulty process executes stage k, and for every correct process q, $m \in (R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_q^k$ at stage k. From Lemma 5.8, for every correct p, no task $stgDeliver_p^k$ remains permanently blocked at line 15, and if task $stgDeliver_p^k$ is terminated, task $stgDeliver_p^{(k+1)}$ is eventually started by p. Thus, all correct processes execute the *when* statement at line 13, and there are two cases two consider: (a) for all $l_k > 0$, every process executes the *then* branch of the *if* statement at line 18 (in which case there are no faulty processes in the system), and (b) for some $l_k > 0$, there is a process r that executes the *else* branch, and R-broadcasts message (k, EndStg).

Case (a). We claim that there exists an $l'_k > 0$ such that $m \in \odot_{\forall r \in \Pi} msgSeq_r^{l'_k}$. From the algorithm, for all r, $msgSeq_r^{l_k} = (R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_r^{k,l_k}$, and so, $m \in msgSeq_r^{l_k}$. Assume, for a contradiction, that for every $l'_k > 0$, $m \notin \odot_{\forall r \in \Pi} msgSeq_r^{l'_k}$. Since $m \in msgSeq_r^{l_k}$, for all r, this can only be possible if for two processes p' and p'', m precedes some message m' in $msgSeq_{p'}^{l_k}$ and m' precedes m in $msgSeq_{p''}^{l_k}$. However, in this case, eventually, $\odot_{\forall r \in \Pi} msgSeq_r = \epsilon$, and processes do not execute the *then* branch, contradicting the assumption of case (a).

Case (b). By the validity of Reliable Broadcast, r R-delivers message (k, EndStg). From Lemma 5.7, if p reaches line 25 in stage k, then q reaches line 25 in stage k, and from agreement of Reliable Broadcast, every correct process q R-delivers (k, EndStg) and executes $propose(k,$

$R\_delivered_q \ominus A\_delivered_q$), such that $m \in R\_delivered_q \ominus A\_delivered_q$. By agreement and termination of Consensus, every q decides on the same $msgStg^k$, and by validity of Consensus $m \in msgStg^k$. It follows that q A-delivers m, a contradiction that concludes the proof. □

Proposition 5.4 (Uniform Integrity). *For any message m, each process A-delivers m at most once, and only if m was previously A-broadcast by $sender(m)$.*

Proof: We first show that, for any message m, each process A-delivers m only if m was previously A-broadcast by $sender(m)$. There are two cases to consider. (a) A process p A-delivers m at line 20. Thus, p received a message $(k, msgSeq_q)$ from every process q, for some k, and $m \in msgSeq_q$. From line 14, $m \in R\_delivered_q$, and from line 12, p has R-delivered m. By uniform integrity of Reliable Broadcast, $sender(m)$ R-broadcasts m, and so, $sender(m)$ A-broadcasts m. (b) Process p A-delivers m at line 30. Thus, from line 29, $m \in msgSet^k$, for some k, and p executed $decide(k, msgStg^k)$. By uniform validity of Consensus, some process q executed $propose(k, R\_delivered_q \ominus A\_delivered_q)$, such that $m \in R\_delivered_q \ominus A\_delivered_q$. From an argument similar to the one presented in item (a), $sender(m)$ A-broadcasts m.

We now show that m is only A-delivered once by p. From Lemma 5.6, it is clear that if m is A-delivered in stage k (i.e., $m \in stgA\_deliver^k \oplus endA\_deliver^k$), then m is not A-delivered in some other stage k', $k' \neq k$. It remains to be shown that m is not A-delivered more than once in stage k. There are three cases to be considered: m is A-delivered at line 20 and will not be A-delivered again (a) at line 20 or (b) at line 30, and (c) m is A-delivered at line 30 and will not be A-delivered again at line 20.

Case (a). After A-delivering m at line 20, p includes m in $stgA\_deliver_p^k$, and from line 19, p will not A-deliver m again at line 20.

Case (b). For a contradiction, assume that m is A-delivered once at line 20 and again at line 30. Thus, when p executes line 29, $m \notin stgA\_deliver_p^k$. Since m has already been A-delivered at line 20, it follows that task $StgDeliver^k$ is terminated after p A-delivers m at line 20 and before p executes line 21. This leads to a contradiction since lines 20 and 21 are executed atomically.

Case (c). Before executing line 30, p executes line 26, and terminates task $StgDeliver^k$. So, once p A-delivers some message at line 30 in stage k, no message can be A-delivered at line 20 in stage k by p. □

Theorem 5.1 *Algorithm 1 solves Atomic Broadcast.*

Proof. Immediate from Propositions 5.1, 5.2, 5.3, and 5.4. □

6 Efficiency of the OPT-ABcast Algorithm

6.1 On the Necessity of Consensus

In this section, we discuss the efficiency of the OPT-ABcast algorithm. Intuitively, the idea is that if Consensus is not needed to deliver some message m, but necessary to deliver some other message m', then the delivery latency of m' is greater than the delivery latency of m. Before going into details about the delivery latency of messages delivered with and without the cost of a Consensus execution (see Section 6.2), we present a more general result about the necessity of Consensus in the OPT-ABcast algorithm. Briefly, Proposition 6.1 states that in a failure free and suspicion free run, Consensus is not executed in stage k if the spontaneous total order property holds in k.

Lemma 6.1 *For any two processes p and q, and all $k \geq 1$, if p executes line 21 for the l_k-th time in stage k, $l_k > 0$, then q executes line 21 for the $(l_k - 1)$-th time in stage k.*

PROOF. If p executes line 21 for the l_k-th time in stage k, then p executes the *wait* statement at line 15 for the l_k-th time in stage k such that p does not suspect any process and receives a message from every process (furthermore, there is a non-empty prefix between all messages received by p). From the no creation property of Reliable Channels, every process q executes the *send* statement at line 14 for the l_k-th time in stage k. For a contradiction, assume that q does not execute line 21 for the $(l_k - 1)$-th time. Then, q executes *R-broadcast*(k, ENDSTG) (line 23) in the l'_k iteration of lines 14-24, $l'_k \leq (l_k - 1)$, and q finishes task *StgDeliver*^{k} (line 24). Therefore, q never executes the *send* statement at line 14 for the l_k-th time, a contradiction. □

Proposition 6.1 *Let R be a failure free and suspicion free run of the OPT-ABcast algorithm. If for every two processes p and q, all $k > 0$, and all $l_k > 0$, $((R\_delivered_p \ominus A\_delivered_p) \ominus stgA\_deliver_p^{k,l_k}) \odot ((R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_q^{k,l_k}) \neq \epsilon$, then no process executes Consensus k in R.*

PROOF. Assume that there is a process p that executes Consensus k in R. From the algorithm, p R-delivers a message of the type (k, ENDSTG), and by uniform integrity of Reliable Broadcast, some process q executed *R-broadcast*(k, ENDSTG). From line 18, either (a) q suspects some process, or (b) there is an iteration $l_k \geq 0$ of lines 14-17, such that $prefix_q^{l_k+1} = \epsilon$. Case (a) contradicts the hypothesis that no process is suspected, so it must be that $prefix_q^{l_k+1} = \epsilon$.

From Lemma 6.1 and lines 17, 14 and 15, $prefix_q^{l_k+1} = \odot_{\forall r} msgSeq_r^{l_k+1} = \odot_{\forall r}((R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_r^{k,l_k})$, and so, $\odot_{\forall r}((R\_delivered_r \ominus A\_delivered_r) \ominus stgA\_deliver_r^{k,l_k}) = \epsilon$. Therefore, there must exist two processes p and q such that $((R\_delivered_p \ominus A\_delivered_p) \ominus stgA\_deliver_p^{k,l_k}) \odot ((R\_delivered_q \ominus A\_delivered_q) \ominus stgA\_deliver_q^{k,l_k}) = \epsilon$, contradicting the hypothesis. □

Thus, from Proposition 6.1, in a failure free and suspicion free run, Consensus is only necessary in stage k if the spontaneous total order property does not hold in k.

6.2 Delivery Latency of the OPT-ABcast Algorithm

We now discuss in more detail the efficiency of the OPT-ABcast algorithm. For every process p and all stages k, there are two cases to consider: (a) messages A-delivered by p during stage k (line 20), and (b) messages A-delivered by p at the end of stage k. The main result is that for case (a), the OPT-ABcast algorithm can A-deliver messages with a delivery latency equal to 2, while for case (b), the delivery latency is at least equal to 4. Since known Atomic Broadcast algorithms deliver messages with a delivery latency of at least 3, these results show the tradeoff of the OPT-ABcast algorithm: if the spontaneous total order property only holds rarely, the OPT-ABcast algorithm is not attractive, while otherwise, the OPT-ABcast algorithm leads to smaller costs than known Atomic Broadcast algorithms.

Propositions 6.2 and 6.3 assess the minimal cost of the OPT-ABcast algorithm to A-deliver a message m. Proposition 6.2 defines a lower bound on the delivery latency of m, and Proposition 6.3 states that this bound can be reached in runs where no process A-delivers m at the end a of stage. We consider a particular implementation of Reliable Broadcast that appears in [4].[4]

Proposition 6.2 *Assume that the OPT-ABcast algorithm uses the Reliable Broadcast implementation presented in [4]. If $\mathcal{R}$ is a set of runs generated by the OPT-ABcast algorithm such that m is a message A-delivered in runs in $\mathcal{R}$, then there is no run R, $R \in \mathcal{R}$, such that $dl^R(m) < 2$.*

PROOF. Assume that m is A-delivered in stage k, and let p be a process that A-delivers m in R. There are two cases to consider: (a) m is A-delivered by p during stage k, and (b) m is A-delivered by p at the end of stage k. In case (a), p received a message $(-, msgSeq_q)$ from every process q such that $m \in msgSet_q$. Since q executes $send(-, R\_delivered_q \ominus A\_delivered_q)$ such that $m \in R\_delivered_q \ominus A\_delivered_q$, q executes *R-deliver*(m), and by uniform integrity of Reliable Broadcast, there is some process r that executes *R-broadcast*(m), which is the process that executes *A-broadcast*(m). From the implementation of Reliable Broadcast, $ts(A\text{-}broadcast_r(m)) = ts(send_r(m))$, and by the definition of delivery latency, $ts(A\text{-}deliver_p(m)) = ts(send_r(m)) + 2$, and so, $dl^R(m) \geq 2$.

In case (b), it follows that p executes *R-deliver*$(-, \text{ENDSTG})$, and so, there is some process q that executes *R-broadcast*$(-, \text{ENDSTG})$ (line 23). Since q executes line 23, it must be that $m \in R\_delivered_q \ominus A\_delivered_q$, and so, q R-delivered m from some r. From an argument similar to the one presented in case (a), $dl^R(m) \geq 2$. □

Proposition 6.3 *Assume that the OPT-ABcast algorithm uses the Reliable Broadcast implementation presented in [4]. If $\mathcal{R}$ is a set of runs generated by the OPT-ABcast algorithm, such that in runs in $\mathcal{R}$, m is a message only A-delivered during stage k, for some $k > 0$, then there is a run R, $R \in \mathcal{R}$, such that $dl^R(m) = 2$.*

PROOF. Immediate from Figure 6, where process p A-broadcasts message m. (Some messages have been omitted from Figure 6 for clarity.) Let $\rho, \rho' \in \{p, q, r, s\}$. We have $ts(receive_\rho(m)) = ts(send_p(m)) + 1$, and $ts(receive_\rho(k, \langle m \rangle)$ from $\rho') = ts(send_{\rho'}(k, \langle m \rangle)) + 1$. But $ts(send_{\rho'}(k, \langle m \rangle)) = ts(receive_{\rho'}(m))$, and therefore, $ts(receive_\rho(k, \langle m \rangle)$ from $\rho') = ts(send_p(m)) + 2$. From Figure 6, we have that $ts(A\text{-}broadcast_p(m)) = ts(send_p(m))$, and $ts(A\text{-}deliver_\rho(m)) = ts(receive_\rho(k, \langle m \rangle)$ from $\rho')$. By the definition of delivery latency, we conclude that $dl^R(m) = 2$. □

The results that follow define the behaviour of the OPT-ABcast algorithm for messages A-delivered at the end of stage k. Proposition 6.4 establishes a lower bound for this case, and Proposition 6.5 shows that this bound can be reached when there are no process failures and no failure suspicions.

[4]Whenever a process p wants to R-broadcast a message m, p sends m to all processes. Once a process q receives m, if $q \neq p$ then q sends m to all processes, and, in any case, q R-delivers m.

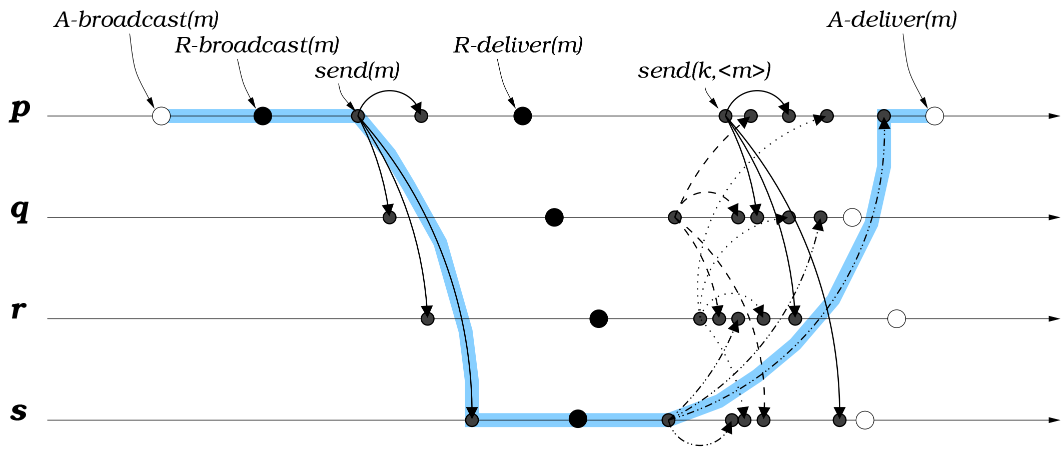

Figure 6: Run of OPT-ABcast with $dl^R(m) = 2$

Proposition 6.4 *Assume that the OPT-ABcast algorithm uses the Reliable Broadcast implementation presented in [4], and the Consensus implementation presented in [14]. Let $\mathcal{R}$ be a set of runs generated by the OPT-ABcast algorithm, such that m and m' are the only messages A-broadcast and A-delivered in $\mathcal{R}$. If m and m' are A-delivered at line 30 by some process p, then there is no run R, $R \in \mathcal{R}$, such that $dl^R(m) < 4$ and $dl^R(m') < 4$.*

PROOF. Assume for a contradiction that there is a run R in $\mathcal{R}$ such that $dl^R(m) < 4$ and $dl^R(m') < 4$. Since p A-delivers m and m' at line 30, p R-delivers message $(-, \text{ENDSTG})$, and by uniform integrity of Reliable Broadcast, there is a process q that executes $R\text{-}broadcast(-, \text{ENDSTG})$. Thus, q has R-delivered at least one message that is neither in $A\_delivered_q$ nor in $stgA\_deliver_q$ (line 13). Without loss of generality, assume that this message is m. Since q R-delivered m, there is a process r that executes $R\text{-}broadcast(m)$, and this is the process that executes $A\text{-}broadcast(m)$. From the definition of delivery latency, we have that $ts(propose_p(-)) = ts(A\text{-}broadcast_r(m)) + 2$. From the contradiction hypothesis, $dl^R(m) = ts(A\text{-}deliver_p(m)) - ts(A\text{-}broadcast_r(m)) < 4$, and so, $ts(A\text{-}deliver_p(m)) = ts(A\text{-}broadcast_r(m)) + 2 + c < 4$, where c is the length of the message chain generated by the Consensus execution (i.e., between $propose_p(-)$ and $decide_p(-)$). We conclude that $c < 2$. This leads to a contradiction since for the Consensus algorithm presented in [14], the minimal messages chain is 2, and therefore, $c \geq 2$. □

Proposition 6.5 *Assume that the OPT-ABcast algorithm uses the Reliable Broadcast implementation presented in [4], and the Consensus implementation presented in [14]. Let $\mathcal{R}$ be a set of runs generated by the OPT-ABcast algorithm, such that in every run in $\mathcal{R}$, m and m' are the only messages A-broadcast and A-delivered, and there are no process failures and no failure suspicions. If m and m' are A-delivered at line 30 by some process p, then there is a run R, $R \in \mathcal{R}$, such that $dl^R(m) = 4$ and $dl^R(m') = 4$.*

PROOF. Immediate from Figure 7, where process q A-broadcasts message m, and process r A-broadcasts message m'. (The Consensus execution and some messages have been omitted for clarity.) For all $\rho \in \{p,q,r,s\}$, $ts(receive_\rho(m)) = ts(send_q(m)) + 1$, and $ts(receive_\rho(m')) = ts(send_r(m')) + 1$. It also follows that $ts(receive_\rho(k, \text{EndStg})) = ts(send_s(k, \text{EndStg})) + 1$. From Figure 7, $ts(send_s(k, \text{EndStg})) = ts(receive_s(m)) = ts(receive_s(m'))$, and therefore, $ts(receive_\rho(k, \text{EndStg})) = ts(send_{\rho'}(m)) + 2$, $\rho' \in \{q, r\}$.

By the Consensus algorithm given in [14], $ts(decide_\rho(-)) = ts(propose_\rho(-)) + 2$. From Figure 7, $ts(propose_\rho(-)) = ts(receive_\rho(k, \text{EndStg}))$, and we have that $ts(decide_\rho(-)) = ts(receive_\rho(k, \text{EndStg})) + 4$. We conclude by the definition of delivery latency, and from $ts(A\text{-}deliver_\rho(m)) = ts(A\text{-}deliver_\rho(m')) = ts(decide_\rho(-))$, $ts(A\text{-}broadcast_q(m)) = ts(send_q(m))$, and $ts(A\text{-}broadcast_r(m)) = ts(send_r(m))$, that $dl^R(m) = 4$ and $dl^R(m') = 4$. □

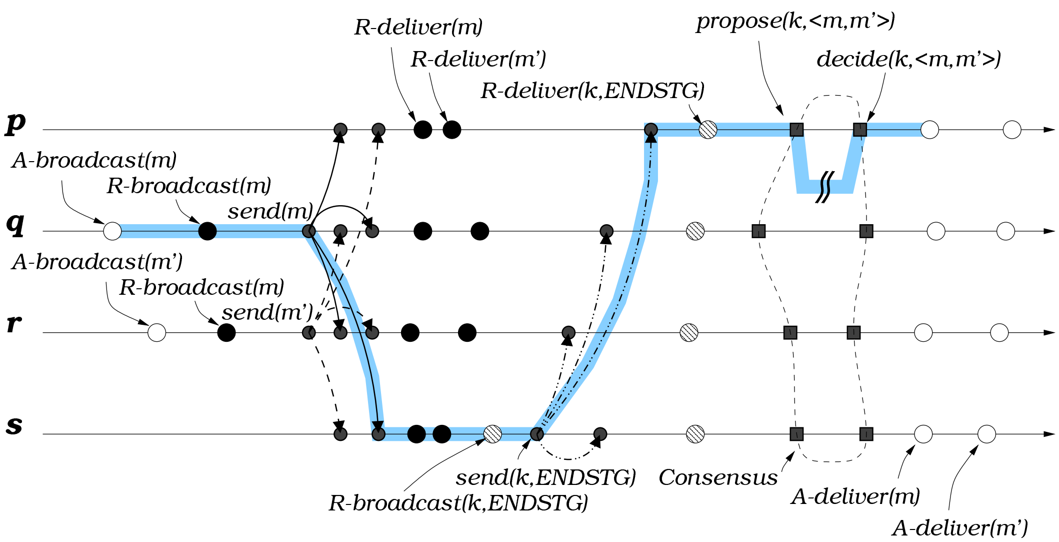

Figure 7: Run of OPT-ABcast with $dl^R(m) = 4$ and $dl^R(m') = 4$

6.3 Cost Analysis of the OPT-ABcast Algorithm

In the following, we characterise the OPT-ABcast algorithm by the number of message exchanged between processes to A-deliver messages during a stage and at the end of a stage. In both cases we consider best case scenarios (i.e., runs in which there are no failures and no suspicions). Moreover, we distinguish the case of a point-to-point network from the case of a broadcast network. In the former case, $(n-1)$ messages are necessary to send a message to all processes, and in the latter case, a sent to all issues only 1 message in the network. Our analysis assumes that the OPT-ABcast algorithm uses the Reliable Broadcast implementation presented in [4], and the Consensus implementation presented in [14].

Messages A-delivered during a stage. A message m that is A-delivered during a stage (see Figure 6) is R-broadcast by $sender(m)$ and R-delivered by all processes. From the implementation of Reliable Broadcast presented in [4], this requires $(n-1)^2$ messages in a point-to-point network, and n messages in a broadcast network. Furthermore, before A-delivering m, every process p receives a message of the type $(k, msgSeq)$, $m \in msgSeq$, from all processes. In a point-to-point network, this requires $n(n-1)$ messages, and in a broadcast network, n messages. Thus, to A-deliver a message during a stage, $(2n-1)(n-1)$ messages are issued in a point-to-point network, and $2n$ messages are issued in a broadcast network.

Messages A-delivered at the end of a stage. A message m that is A-delivered at the end of a stage (see Figure 7) has the cost of a message A-delivered during a stage plus the cost of a Consensus execution. From the Consensus implementation given in [14], in a point-to-point network, $2n(n-1)$ messages are issued, and in a broadcast network, $2n-1$ messages are issued. Therefore, to A-deliver a message at the end of a stage, $(4n-1)(n-1)$ messages are issued in a point-to-point network, and $4n-1$ messages are issued in a broadcast network.

The OPT-ABcast in perspective. As a reference, Chandra and Toueg Atomic Broadcast algorithm, based on the Reliable Broadcast implementation presented in [4], and the Consensus implementation presented in [14], issues $(3n-1)(n-1)$ messages to deliver a message in a point-to-point network, and $3n-1$ messages in a broadcast network. Table 1 shows the results presented in this section and in the previous section.

Algorithm	**point-to-point**		**broadcast**		**delivery latency**
OPT-ABcast(during a stage)	$(2n-1)(n-1)$	$O(n^2)$	$2n$	$O(n)$	2
OPT-ABcast(end of a stage)	$(4n-1)(n-1)$	$O(n^2)$	$4n-1$	$O(n)$	4
CT Atomic Broadcast [4]	$(3n-1)(n-1)$	$O(n^2)$	$3n-1$	$O(n)$	3

Table 1: Cost of delivering a message

7 Handling Failures

In the OPT-ABcast algorithm (line 18), whenever task $StgDeliver^k$ does not receive messages from all processes in Π, the current stage k is terminated, which leads to an execution of Consensus to A-deliver the messages. Therefore, as soon as a process $p \in \Pi$ crashes, the A-deliver of messages will always be *slow* (i.e., with a delivery latency of at least 4). This can easily be solved by adding a membership service to our OPT-ABcast algorithm as follows. Let v_i be the current view of system Π ($v_i \subseteq \Pi$):

- at line 18, replace condition $\pi = \Pi$ by $\pi = v_i$.

Once a process p crashes (or is suspected to have crashed), p is removed from the view, and *fast* A-deliver of messages is again possible. We do not discuss further this extension to the OPT-ABcast algorithm, but we note that the instance of the membership problem needed to remove a crashed process can easily be integrated into the Consensus problem that terminates a stage.

8 Conclusion

This work originated from the pragmatic observation that, with high probability, messages broadcast in a local area network are "spontaneously" totally ordered. Exploiting this observation led us to develop the OPT-ABcast algorithm. Processes executing the OPT-ABcast algorithm progress in a sequence of stages, and messages can be delivered during stages or at the end of stages. Messages are delivered faster during stages than at the end of stages. For any process, the current stage is terminated, and another one started, whenever the spontaneous total order property does not hold.

The efficiency of the OPT-ABcast algorithm has been quantified using the notion of delivery latency. The delivery latency of messages delivered during a certain stage has been shown to be equal to 2 (best case), while the delivery latency of messages delivered at the end of a stage equal to 4 (best case). This result shows the tradeoff of the OPT-ABcast algorithm: if most messages are delivered during the stages, the OPT-ABcast algorithm outperforms known Atomic Broadcast algorithms, otherwise, the OPT-ABcast algorithm is outperformed by known Atomic Broadcast algorithms.

Finally, to the best of our knowledge, the OPT-ABcast algorithm is the first agreement algorithm to exploit an optimistic property. If this property is satisfied the efficiency of the algorithm is improved, if the property is not not satisfied the efficiency of the algorithm deteriorates (however the optimistic property has no impact on the safety and liveness guarantees of the system). We believe that this opens interesting perspectives for revisiting or improving other agreement algorithms.

References

[1] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. Number 13, pages 551–560, May 1993.

[2] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[5] J. M. Chang and N. Maxemchuck. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[6] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[7] H. Garcia-Molina and A. Spauster. Ordered and Reliable Multicast Communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991.

[8] R. Guerraoui, M. Larrea, and A. Schiper. Reducing the cost for non-blocking in atomic commitment. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 692–697, May 1996.

[9] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. ACM Press/Addison-Wesley, second edition, 1993.

[10] P. Jalote. Efficient ordered broadcasting in reliable csma/cd networks. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 112–119, May 1998.

[11] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

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

[13] S. W. Luan and V. D. Gligor. A Fault-Tolerant Protocol for Atomic Broadcast. *IEEE Trans. Parallel & Distributed Syst.*, 1(3):271–285, July 90.

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

[15] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

[16] U. Wilhelm and A. Schiper. A Hierarchy of Totally Ordered Multicasts. In *14th IEEE Symp. on Reliable Distributed Systems (SRDS-14)*, pages 106–115, Bad Neuenahr, Germany, September 1995.