Technical Report

UCAM-CL-TR-671
ISSN 1476-2986

Number 671

Low-latency Atomic Broadcast in the presence of contention

Piotr Zieliński

July 2006

ISSN 1476-2986

Low-latency Atomic Broadcast in the presence of contention

Piotr Zieliński

University of Cambridge
Cavendish Laboratory

piotr.zielinski@cl.cam.ac.uk

Abstract

The Atomic Broadcast algorithm described in this paper can deliver messages in two communication steps, even if multiple processes broadcast at the same time. It tags all broadcast messages with the local real time, and delivers all messages in order of these timestamps. The Ω-elected leader simulates processes it suspects to have crashed ($\Diamond S$). For fault-tolerance, it uses a new cheap Generic Broadcast algorithm that requires only a majority of correct processes ($n > 2f$) and, in failure-free runs, delivers all non-conflicting messages in two steps. The main algorithm satisfies several new lower bounds, which are proved in this paper.

1 Introduction

In a distributed system, messages broadcast by different processes at approximately the same time might be *received* by other processes in different orders. Atomic Broadcast is a fault-tolerant primitive, usually implemented on top of ordinary broadcast, which ensures that all processes *deliver* messages to the user in the same order. Applications of Atomic Broadcast include state machine replication, distributed databases, distributed shared memory, and others [4].

As opposed to ordinary broadcast, Atomic Broadcast requires multiple communication steps, even in runs without failures. One of the goals in broadcast protocol design is minimizing the latency in common, failure-free runs, while possibly allowing worse performance in runs with failures. This paper presents an algorithm that is faster, in this respect, than any previously proposed one, and requires only two communication steps, even if multiple processes broadcast at the same time.

The definition of latency in this context can be a source of confusion. In this paper, latency is the time between the atomic broadcast (*abcast*) of a message and its atomic delivery. Note that some papers [5, 8] ignore the step in which the sender physically broadcasts the message to other processes; in that case one step must be added to the reported latency figure.

The algorithm presented here assumes an asynchronous system with a majority of correct processes and the $\Diamond S$ failure detector [2]. Motivated by the increasing availability

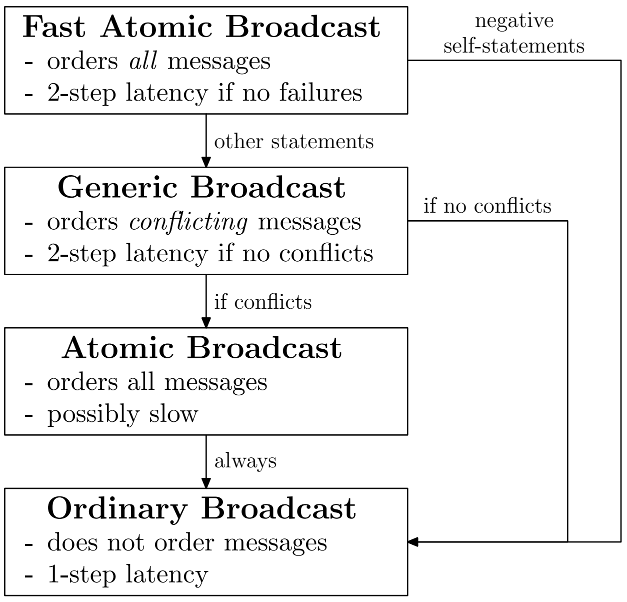


Figure 1: Layered structure of the Atomic Broadcast algorithm presented in this paper.

of services such as GPS or Network Time Protocol [9], I additionally assume that each process is equipped with a (possibly inaccurate) real-time clock. The optimum latency of two steps is achieved if the clocks are synchronized, and degrades gracefully otherwise. In particular, no safety or liveness properties depend on the clock accuracy.

The algorithm employs a well-known method first proposed by Lamport [7]: senders independently timestamp their messages, which are then delivered in the order of these timestamps. The novelty of my approach consists of using real-time clocks in conjunction with unreliable failure detectors [2] and Generic Broadcast [1, 12] to ensure low latency and fault-tolerance at the same time.

The layered structure of the algorithm is shown in Fig. 1. Each process tags all its abcast messages with local real time, and disseminates this information. Both positive and negative *statements* are used ("m abcast at time 51" vs. "no messages abcast between times 30 and 50"). To achieve fault-tolerance, the Ω-elected leader occasionally broadcasts negative statements on behalf of processes it suspects to have crashed ($\Diamond S$). Generic Broadcast is used to resolve conflicts that might occur if the leader's suspicions are wrong. Any Generic Broadcast algorithm can be used here, however, the new algorithm in Sect. 5 is specifically tailored to this situation. It requires only a majority of correct processes ($n > 2f$) and, in failure-free runs, delivers all non-conflicting messages in two communication steps.

This paper is structured as follows. Section 2 formalizes the system model and gives a precise definition of Atomic Broadcast. Section 3 presents the main ideas, which are then transformed into an algorithm in Sect. 4. Section 5 describes a new Generic Broadcast algorithm, that is optimized for the use by the main algorithm. Section 6 shortly discusses some secondary properties of the algorithm. Section 7 presents three new lower bounds that prove two-step delivery cannot be maintained if the system assumptions are relaxed (for example, by eliminating real-time clocks). Section 8 concludes the paper.

1.1 Related Work

A run of an algorithm is *good* if the real-time clocks are accurate and there are no failures or suspicions. In such runs, the algorithm presented here delivers all messages in two steps. No such algorithm has been proposed before; out of over fifty Atomic Broadcast protocols surveyed by Défago et al. [4], the only indulgent algorithm capable of delivering all messages faster than in three steps was proposed by Vicente and Rodrigues [13]. It achieves a latency of $2d+\delta$, where d is the single message delay and $\delta > 0$ is an arbitrarily small constant. The price for having a very small δ is high network traffic; the number of messages is proportional to $1/\delta$. In comparison, my algorithm achieves the latency of $2d$ with the number of messages dependent only on the number of processes.

Several broadcast protocols achieve a two-step latency in some good runs, such as those with all messages spontaneously received in order (Optimistic Atomic Broadcast [10, 14]) or those without conflicting messages (Generic Broadcast [1, 10, 12, 14]). In comparison, the algorithm presented in this paper delivers messages in two steps in all good runs.

Défago et al. [4] proposed a classification scheme for broadcast algorithms. According to that scheme, the algorithm described in this paper is a time-based communication-history algorithm for closed groups, similarly to the original Lamport's algorithm [7], which motivated it.

2 System Model and Definitions

The system model consists of n processes $p_1, \ldots, p_n$, out of which at most f can fail by crashing. Less than a half of all the processes are faulty ($n > 2f$). Processes communicate through asynchronous reliable channels, that is, there is no time limit on message transmission time, and messages between correct processes never get lost.

Each process is equipped with: (i) a possibly inaccurate, non-decreasing real-time clock, (ii) an unreliable leader oracle Ω, which eventually outputs the same correct leader at all correct processes, and (iii) a failure detector $\Diamond S$. Failure detector $\Diamond S$ outputs a list of processes that it suspect to have crashed. It ensures that (i) all crashed processes will eventually be suspected by all correct processes, and (ii) at least one correct process will eventually never be suspected by any correct process [3]. Detector $\Diamond S$ is the weakest suspicion-list-like failure detector that makes Atomic Broadcast solvable in asynchronous settings. It can implement Ω [3], so the Ω assumption can, technically, be dropped.

In Atomic Broadcast, processes abcast messages, which are then delivered by all processes in the same order. Formally [6],

Validity. If a correct process abcasts a message m, then all correct processes will eventually deliver m.

Uniform Agreement. If a process delivers a message m, then all correct processes eventually deliver m.

Uniform Integrity. For any message m, every process delivers m at most once, and only if m was previously abcast.

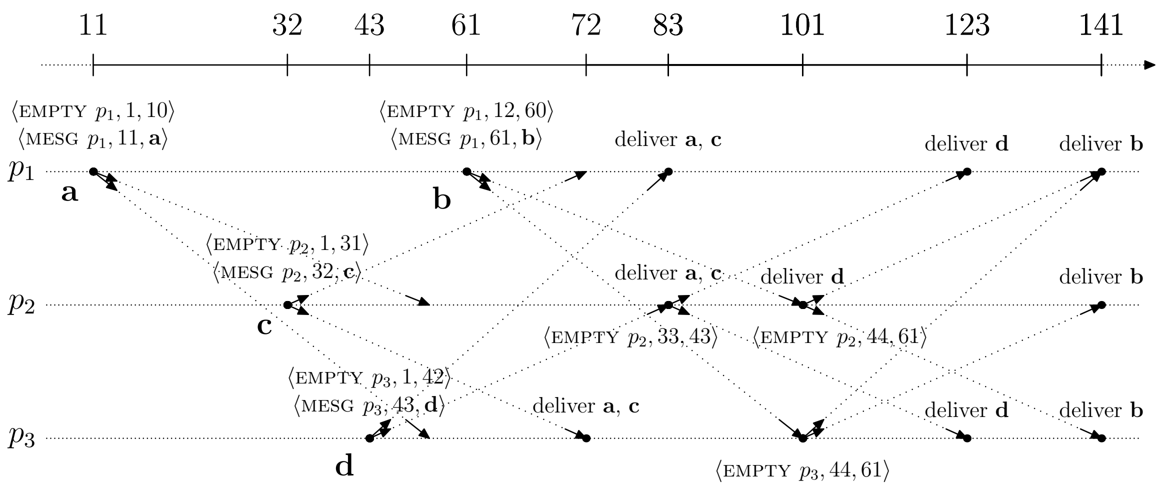


Figure 2: A run that uses ordinary broadcast for all statements (not fault-tolerant). One step is 40 units of time. The fault-tolerant version is shown in Fig. 4.

Uniform Total Order. If some process delivers message m' after message m, then every process delivers m' only after it has delivered m.

I measure latency in *communication steps*, where one communication step is the maximum message delay d between *correct* processes (possibly ∞). Processes do not know d. In good runs, the algorithm described in this paper delivers all messages in two communication steps ($2d$), regardless of the number of processes abcasting simultaneously. In other runs, the performance can be worse, however, the above four properties of Atomic Broadcast always hold (Sect. 6).

3 Atomic Broadcast Algorithm

The algorithm employs a well-known method proposed by Lamport [7]: senders independently timestamp their messages, which are then delivered in the order of these timestamps. In the scenario in Fig. 2, messages **a**, **b**, **c**, **d** are tagged by their senders with timestamps 11, 61, 32, and 43. As a result, they should be delivered in the order: **a**, **c**, **d**, and **b**.[1]

3.1 Runs without Failures

How can we implement this idea is a message-passing environment? To deliver messages in the right order, a process, say p_3, must know the timestamps of messages **a**, **b**, **c**, **d**, and that no other messages were abcast. Let the *abcast state* of p_i at time t be the message (if any) abcast by p_i at time t. For example, the abcast state of p_2 at time 32 is **c**, and empty at time 34.

Processes share information by broadcasting their abcast states. When a process p_i abcasts a message m at time t, it broadcasts two statements (as separate messages):

[1]For simplicity, I assume that timestamps are (possibly very large) integers whose last digit is the process number, so that no two messages can carry the same timestamp.

1. A *positive* statement $\langle \textsc{mesg}\ p_i, t, m\rangle$ saying that p_i abcast message m at time t.
2. A *negative* statement $\langle \textsc{empty}\ p_i, t'+1, t-1\rangle$ saying that p_i abcast no messages since abcasting its previous message at time t' ($t' = 0$ if m is the first message abcast by p_i).

In the example in Fig. 2, the following statements are broadcast:

$$\langle \textsc{empty}\ p_1, 1, 10\rangle,\quad \langle \textsc{mesg}\ p_1, 11, \mathbf{a}\rangle,\quad \langle \textsc{empty}\ p_1, 12, 60\rangle,\quad \langle \textsc{mesg}\ p_1, 61, \mathbf{b}\rangle,$$
$$\langle \textsc{empty}\ p_2, 1, 31\rangle,\quad \langle \textsc{mesg}\ p_2, 32, \mathbf{c}\rangle,\quad \langle \textsc{empty}\ p_3, 1, 42\rangle,\quad \langle \textsc{mesg}\ p_3, 43, \mathbf{d}\rangle.$$

After receiving these statements, we have complete information about all processes abcast states up to time 32. We can deliver messages **a** and **c**, in this order, because we know that no other messages were abcast with a timestamp ≤ 32. On the other hand, **d** cannot be delivered yet, because we do not have any information about the abcast state of process p_2 after time 32. If we delivered **d**, and later found out that p_2 abcast **e** at time 42, we would violate the rule that messages are delivered in order of their timestamps ($43 \not< 42$).

To deliver **d**, we need p_2's help. When p_2 learns that p_3 abcast a message at time 43, it announces its abcast states by broadcasting $\langle \textsc{empty}\ p_2, 33, 43\rangle$ (Fig. 2). Similarly, when p_2 and p_3 learn about **b**, they broadcast $\langle \textsc{empty}\ p_2, 44, 61\rangle$ and $\langle \textsc{empty}\ p_3, 44, 61\rangle$ respectively. Note that p_1 does not need to broadcast anything, because by the time it learnt about **c** and **d**, it had already broadcast the necessary information while abcasting **b**. After receiving all these statements, we now have complete information about all processes abcast states up to time 61. In addition to previously delivered **a** and **c**, we can now deliver **d** and **b** as well. Note that, in the failure-free case, the order in which processes receive statements is irrelevant.

3.2 Dealing with Failures

What would happen if p_2 crashed immediately after sending **c**? Process p_2 would never broadcast the negative statement $\langle \textsc{empty}\ p_2, 33, 43\rangle$, so the algorithm would never deliver **d** and **b**.

To cope with this problem, the current leader (Ω) broadcasts the required negative statements on behalf of all processes it suspects ($\Diamond S$) to have failed. For example, if p_1 is the leader, suspects p_2, and learns that p_3 abcast **d** at time 43, then p_1 broadcasts $\langle \textsc{empty}\ p_2, 1, 43\rangle$. This allows message **d** to be delivered.

Allowing p_1 to make negative statements on behalf of p_2 opens a whole can of worms. To start with, $\langle \textsc{empty}\ p_2, 1, 43\rangle$ blatantly contradicts $\langle \textsc{mesg}\ p_2, 32, \mathbf{b}\rangle$ broadcast earlier by p_2. A similar conflict occurs if p_1 is wrong in suspecting p_2, and p_2 decides to abcast another message **e**, say at time 42. In general, two statements *conflict* if they carry different information about the abcast state of the same process at the same time.

The problem of conflicting statements can be solved by assuming that, if a process receives two conflicting statements, then the first one wins. For example, receiving

$$\langle \textsc{mesg}\ p_2, 32, \mathbf{b}\rangle, \langle \textsc{empty}\ p_2, 1, 43\rangle, \langle \textsc{mesg}\ p_2, 42, \mathbf{e}\rangle$$

is equivalent to

$$\langle \textsc{mesg}\ p_2, 32, \mathbf{b}\rangle, \langle \textsc{empty}\ p_2, 33, 43\rangle.$$

We can ensure that all processes receive all conflicting statements in the same order by using Generic Broadcast [1, 12] to broadcast them. Unlike Atomic Broadcast, Generic Broadcast imposes order only on conflicting messages, which leads to good performance in runs without conflicts.

3.3 Latency Considerations

In order to achieve a two-step latency in good runs, all positive statements must be delivered in two communication steps. Negative statements are even more problematic, because they may be issued one step after the abcast event that triggered them (e.g., $\langle \textsc{empty}\ p_2, 33, 43\rangle$ triggered by p_3 abcasting **d** at time 43 in Fig. 2). Therefore, negative statement must be delivered in at most one step. The following observations show how to satisfy these requirements.

Observation 1: No conflicts in good runs

Good runs have no suspicions, so processes issue statements only about themselves. These *self-statements* never conflict. Since no conflicting statements are issued, Generic Broadcast will deliver all statements in two communication steps [1, 12].

Observation 2: No conflicts involving negative self-statements

Statements made by processes can be divided into three groups: positive self-statements, negative self-statements, and negative statements made by the leader. Negative self-statements do not conflict with any of these because (i) self-statements do not conflict because they talk about different processes or times, and (ii) negative statements do not conflict because they carry the same information "no messages". Therefore, negative self-statements do not require Generic Broadcast; ordinary broadcast, which takes only one communication step, is sufficient.

4 Implementation

Figure 3 presents the details of the algorithm sketched in Sect. 3. It can be conceptually divided into two parts: broadcasting (lines 1–13) and delivery (lines 14–28).

4.1 Broadcasting Part (Lines 1–13)

Each process p_i maintains two variables: $time_i$ and $t_i^{\max}$. The read-only variable $time_i$ is an integer representing the current reading of p_i's clock. Its value increases at the end of every "block" (e.g. lines 2–6), but remains constant within it. Variable $t_i^{\max}$ represents the highest time for which p_i broadcast a statement about its abcast state. For example $t_1^{\max} = 61$ after p_1 abcast **a** and **b** (Fig. 4). Initially $t_i^{\max} = 0$.

To abcast a message m, a process p_i first broadcasts $\langle \textsc{active}\ time_i\rangle$, which informs other processes that some message was abcast at time $time_i$. Then, p_i broadcasts one negative and one positive statement using ordinary and Generic Broadcast, respectively. They inform other processes that p_i abcast m at time $time_i$, and nothing between $t_i^{\max} + 1$ and $time_i - 1$. Finally, p_i updates $t_i^{\max}$.

```
$t_i^{\max} \leftarrow 0$                                   { *the highest timestamp used so far* }
**when** $p_i$ executes $abcast(m)$ **do**
  broadcast $\langle$ACTIVE $time_i\rangle$ using ordinary broadcast
  broadcast $\langle$EMPTY $p_i, t_{\max}+1, time_i - 1\rangle$ using ordinary broadcast
  broadcast $\langle$MESG $p_i, time_i, m\rangle$ using Generic Broadcast
  $t_i^{\max} \leftarrow time_i$                            { *$time_i$ increases after this line* }
**when** $p_i$ received $\langle$ACTIVE $t\rangle$ in the past and $t_i^{\max} < t \leq time_i$ **do**
  broadcast $\langle$EMPTY $p_i, t_i^{\max}+1, t\rangle$ using ordinary broadcast
  $t_i^{\max} \leftarrow t$                                 { *$time_i$ increases after this line* }
**when** change in $t_i^{\max}$ or the output of the failure detector or leader oracle **do**
  **if** $p_i$ considers itself a leader
    **for** all suspected processes $p_j \neq p_i$ **do**
      broadcast $\langle$EMPTY $p_j, 1, time_i\rangle$ using Generic Broadcast
**task** delivery at process $p_i$ **is**
  $todeliver_i \leftarrow \emptyset$; $known_i[j] \leftarrow \emptyset$ for all $j = 1, \ldots, n$
  **repeat forever**
    **wait for** a statement delivered by ordinary or Generic Broadcast
      **if** negative statement $\langle$EMPTY $p_j, t_1, t_2\rangle$ delivered **then**
        $known_i[j] \leftarrow known_i[j] \cup [t_1, t_2]$
      **if** positive statement $\langle$MESG $p_j, t, m\rangle$ delivered **then**
        **if** $t \notin known_i[j]$ **then**
          send $\langle$ACTIVE $t\rangle$ to itself
          add $(m, t)$ to $todeliver_i$
          $known_i[j] \leftarrow known_i[j] \cup \{t\}$
        **else if** $p_i = p_j$ **then** $abcast(m)$         { *the sender tries again* }
    **let** $t_i^{\text{known}} = \max\{\, t \mid [1, t] \subseteq known_i[j]$ for all $j\,\}$
    **for** all $(m, t) \in todeliver_i$ with $t \leq t_i^{\text{known}}$, in the order of increasing $t$ **do**
      atomically deliver $m$; remove $(m, t)$ from $todeliver_i$
```

Figure 3: Atomic Broadcast algorithm with a two-step latency in good runs. It requires the $\Diamond P$ failure detector; see Sect. 4.3 for the modifications required for $\Diamond S$.

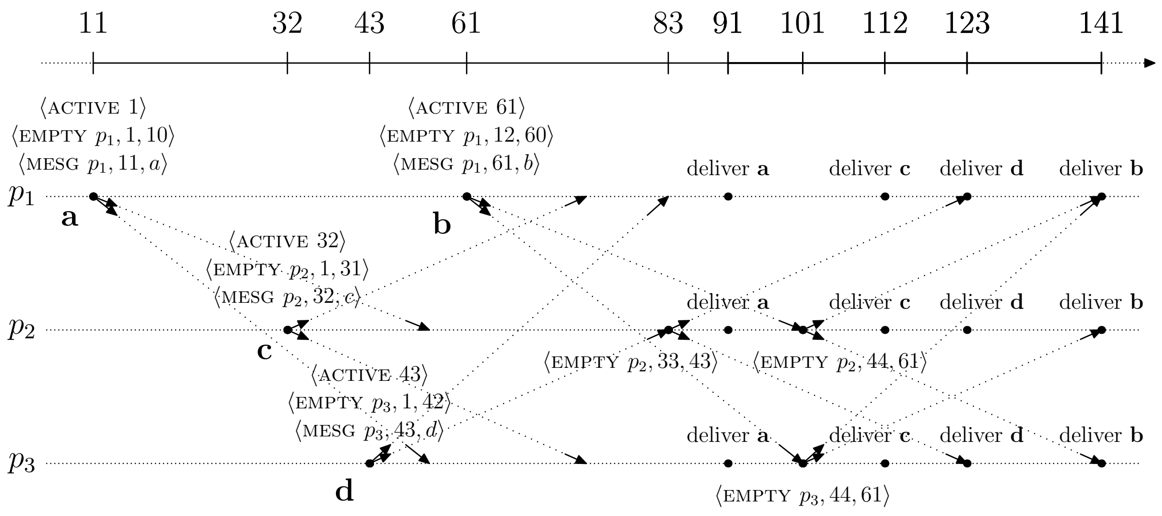


Figure 4: A example run of the fault-tolerant algorithm from Fig. 3. Generic Broadcast of statements ⟨MESG⟩ takes two steps; the related messages are not shown.

When p_i receives ⟨ACTIVE t⟩ with $t_i^{\max} < t \leq \mathit{time}_i$, it broadcasts ⟨EMPTY $p_i, t_i^{\max}{+}1, t$⟩ and updates $t_i^{\max}$. This informs other processes that p_i abcast no messages between $t^{\max}{+}1$ and t. If $t_i^{\max} \geq t$, then p_i has already reported its abcast states for time t and before, so no new statement is needed.

The condition $t \leq \mathit{time}_i$ makes sure that $t_i^{\max} < \mathit{time}_i$ at all times, so that the abcast function does not a issue a conflicting self-statement in line 5. This condition always holds in good runs: $t > \mathit{time}_i$ would mean that the message ⟨ACTIVE t⟩ arrived from a process whose clock was at least one communication step ahead of p_i's. In this case, p_i simply waits until $t \leq \mathit{time}_i$ holds, and then executes lines 7–9.

Lines 10–13 are executed when a leader process p_i experiences a change in $t_i^{\max}$ or in the output of its failure detector $\Diamond S$ or the leader oracle Ω. In these cases, p_i issues the appropriate negative statements on behalf of all processes it suspects to have crashed.

4.2 Delivery Part (Lines 14–28)

The delivery tasks delivers messages in the order of their timestamps. Each process p_i maintains two variables: $\mathit{todeliver}_i$ and known_i. Variable $\mathit{todeliver}_i$ contains all timestamped messages (m, t) that have been received but not atomically delivered yet. For example, at time say 80, $\mathit{todeliver}_2 = \{(a, 11), (c, 32)\}$ (Fig. 4).

Variable known_i is an array of sets. Each $\mathit{known}_i[j]$ is the set of all times for which the abcast state of p_j is known, initially $\emptyset$. For example, at time 80, $\mathit{known}_2[1] = [1, 10]$, where $[t_1, t_2] = \{\, t \mid t_1 \leq t \leq t_2 \,\}$. Time $11 \notin \mathit{known}_2$ because ⟨MESG $p_1, 11, \mathbf{a}$⟩, sent using Generic Broadcast, will arrive at p_2 two communication steps after being sent, that is, at time 91. Each set $\mathit{known}_i[j]$ can be compactly represented as union of intervals $[t_1, t_2]$; most of the time $\mathit{known}_i[j] = [1, t]$ for some t.

In each iteration of the infinite loop (lines 16–28), the delivery task waits for a statement delivered by ordinary or Generic Broadcast. After receiving ⟨EMPTY p_j, t_1, t_2⟩, process p_i updates its knowledge about p_j by adding the interval $[t_1, t_2]$ to $\mathit{known}_i[j]$. For ⟨MESG p_j, t, m⟩, p_i first checks whether it has received any information about the abcast state of p_j at time t before. If not, p_i schedules message m for delivery by adding the pair (m, t) to $todeliver_i$. Process p_i also adds $\{t\}$ to $known_i[j]$ to reflect its knowledge of the abcast state of p_j at time t. Sending $\langle \textsc{active}\ t\rangle$ in line 22 is necessary to ensure Uniform Agreement on messages abcast by faulty processes. Messages $\langle \textsc{active}\ t\rangle$ broadcast by such processes can get lost, so line 22 serves as a backup.

If $t \in known_i[j]$, then the leader suspected p_j to have crashed, and broadcast a conflicting negative statement on p_j's behalf, which was delivered before $\langle \textsc{mesg}\ p_j, t, m\rangle$. Since message m cannot be delivered with timestamp t, its sender tries to abcast it again, with a new timestamp. In the future, if m cannot be delivered with the new timestamp, its sender will re-abcast it yet again, and so on. If the failure detector is $\Diamond P$, all correct processes will eventually be permanently not suspected, so some re-abcast of m will eventually result in delivering m. For dealing with $\Diamond S$, see Sect. 4.3.

After processing the statement received in line 17, p_i attempts to deliver abcast messages. It first computes the largest time t_i^{known} for which it knows the abcast states of all processes up to time t_i^{known}. This ensures that $todeliver_i$ contains all non-yet-delivered messages abcast by time t_i^{known}. Process p_i delivers all these messages in the order of increasing timestamps and removes them from $todeliver_i$.

4.3 Dealing with $\Diamond S$ by Leader-Controlled Retransmission

With the $\Diamond S$ failure detector, the algorithm from Fig. 3 might fail to deliver messages abcast by senders that are correct but permanently suspected by the leader (this cannot happen with $\Diamond P$). This problem can be solved by letting the leader re-abcast all messages, instead of each process doing so itself.

In this scheme, each process p_i maintains a set $\mathcal{B}_i$ of messages it abcast but not delivered yet. Periodically, it sends $\mathcal{B}_i$ to the current leader, who re-abcasts all $m \in \mathcal{B}_i$ on p_i's behalf. Since Ω guarantees an eventual stable leader, each message abcast by a correct p_i will eventually be delivered (Validity). Some messages might be delivered twice, so an explicit duplicate elimination must be employed.

5 Cheap Generic Broadcast

The Atomic Broadcast algorithm from Sect. 4 assumes that, in failure-free runs, the underlying Generic Broadcast delivers all non-conflicting messages in two communication steps. Achieving this with existing Generic Broadcast algorithms requires $n > 3f$ [1, 12, 14].

Figure 5 presents a Generic Broadcast algorithm, which is similar to [1] but requires only $n > 2f$ (cheapness). As opposed to other Generic Broadcast algorithms, it achieves a two-step latency only in failure-free runs (otherwise $n > 3f$ would be a lower bound [11]). As a bonus, all non-conflicting messages are delivered in three steps even in runs with failures, so the algorithm in Fig. 5 could be seen as a generalization of three-step Generic Broadcast protocols that require $n > 2f$ [1, 12]. To deal with conflicting messages, the algorithm employs an auxiliary Atomic Broadcast protocol, such as [2]. Real-time clocks are not used.

To execute *gbcast*(m), the sender sends $\langle \textsc{first}\ m\rangle$ using Reliable Broadcast [6]. When a process p_i receives $\langle \textsc{first}\ m\rangle$, it first checks whether any messages conflicting with m

```
$seen_i^1 \leftarrow \emptyset$; $seen_i^2 \leftarrow \emptyset$; $quick_i \leftarrow \emptyset$
when $p_i$ executes $gbcast(m)$ do                    { broadcast $m$ using Generic Broadcast }
  broadcast $\langle$FIRST $m\rangle$ using (non-uniform) Reliable Broadcast
when $p_i$ receives $\langle$FIRST $m\rangle$ do
  add $m$ to $seen_i^1$
  if $seen_i^1 \cap C(m) = \emptyset$               { $C(m)$ is the set of messages conflicting with $m$ }
    then FIFO broadcast $\langle$SECOND good $m\rangle$
    else FIFO broadcast $\langle$SECOND bad $m\rangle$
when $p_i$ receives $\langle$SECOND good $m\rangle$ from all processes do
  deliver $m$ if not delivered already                              { two-step delivery }
when $p_i$ receives $\langle$SECOND $*$ $m\rangle$ from $n-f$ processes do
  add $m$ to $seen_i^2$
  if all "*" are "good" and $seen_i^2 \cap C(m) = \emptyset$
    then add $m$ to $quick_i$; broadcast $\langle$THIRD good $m$ $\emptyset\rangle$
    else broadcast $\langle$THIRD bad $m$ $conflicts_i\rangle$ where $conflicts_i = quick_i \cap C(m)$
when $p_i$ receives $\langle$THIRD $*$ $m$ $conflicts_j\rangle$ from $n-f$ processes $p_j$ do
  if all "*" are "good"
    then deliver $m$ if not delivered already                      { three-step delivery }
    else atomically broadcast $\langle$ATOMIC $m$ $conflicts\rangle$ with $conflicts = \bigcup_j conflicts_j$
when a process atomically delivers $\langle$ATOMIC $m$ $conflicts\rangle$ do
  deliver all $m' \in conflicts$ if not delivered already
  deliver $m$ if not delivered already
```

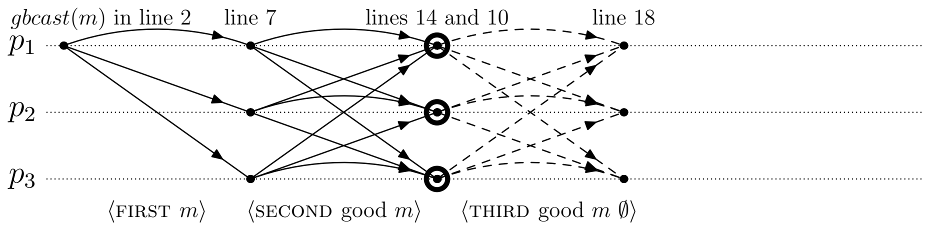
Figure 5: Generic Broadcast algorithm that achieves a two-step latency in good runs and requires only $n > 2f$ (cheapness).

have reached this stage before. ($C(m)$ is the set of messages conflicting with m.) Process p_i then broadcasts $\langle$SECOND good $m\rangle$ or $\langle$SECOND bad $m\rangle$ accordingly. In failure-free runs, p_i receives $\langle$SECOND good $m\rangle$ from all processes, and delivers m immediately (two steps in total, Fig. 6a).

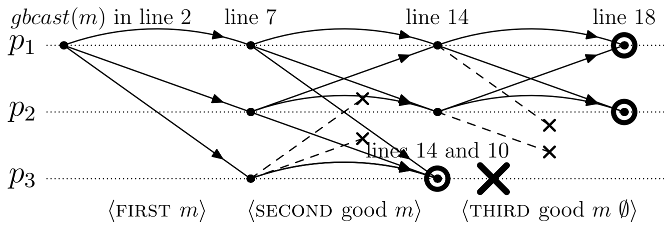
When process p_i receives $n - f$ $\langle$SECOND $*$ $m\rangle$ messages, it checks whether all of them are "good" and no conflicting m' has reached this stage before. The appropriate $\langle$THIRD good m $*\rangle$ or $\langle$THIRD bad m $*\rangle$ message is broadcast. In the "good" case, p_i adds m to $quick_i$, the set of messages that *may* be delivered without using the underlying Atomic Broadcast protocol. In the "bad" case, the $\langle$THIRD$\rangle$ message also contain the set of "quick" messages conflicting with m.

When p_i receives $n-f$ messages $\langle$THIRD good m $\emptyset\rangle$, it delivers m straight away (three steps in total, Fig. 6b). Thus, if m is delivered in this way, $m \in quick_j$ for at least $n - f$ processes p_j. As a result, for any $m' \in C(m)$, all messages $\langle$THIRD $*$ m' $conflicts_j\rangle$ broadcast by these processes have $m \in conflicts_j$. Since $n > 2f$, any two groups of $n - f$ processes overlap, so any process p_i receiving $n-f$ messages $\langle$THIRD $*$ m' $conflicts_j\rangle$, will have $m \in conflicts_j$ for at least one of them.

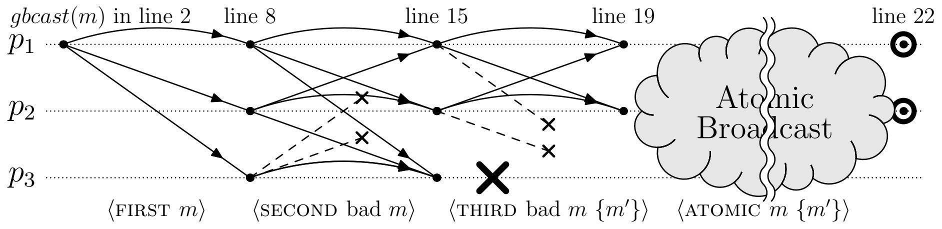
When p_i receives $n-f$ messages $\langle$THIRD $*$ m' $conflicts_j\rangle$, not all "good", it atomically



(a) No conflicts, no failures: delivery in 2 steps. Line 14 gets executed before line 10, but this order does not matter.



(b) No conflicts, one failure: delivery in 3 steps. Note that some processes can deliver earlier than others.



(c) Message m conflicts with a previously gbcast m', the latency depends on the underlying Atomic Broadcast protocol.

Figure 6: Several runs of the Generic Broadcast protocol from Fig. 5 with $n = 3$ and $f = 1$. The apparent synchrony in the diagrams is only for illustrative purposes.

broadcasts m' along with the union *conflicts* of all $n - f$ received sets $\mathit{conflicts}_j$. As explained above, *conflicts* contains all messages $m \in C(m')$ that might be delivered in lines 10 or 18. Processes deliver m' in line 22 only after delivering all $m \in$ *conflicts* in line 21. Otherwise, different processes could deliver m' (line 22) and messages $m \in$ *conflicts* (line 18) in different orders.

In conflict-free runs, no "bad" messages are sent, so all messages are delivered in at most three steps (Figs. 6ab). Figure 6c shows that runs with conflicting messages can have a higher latency, which depends on the latency of the underlying Atomic Broadcast. Finally, observe that FIFO broadcast used for ⟨SECOND⟩ messages ensures that, if m was delivered in line 10, no $m' \in C(m)$ will reach line 12 before m, so m will be delivered in line 18 by all correct processes.

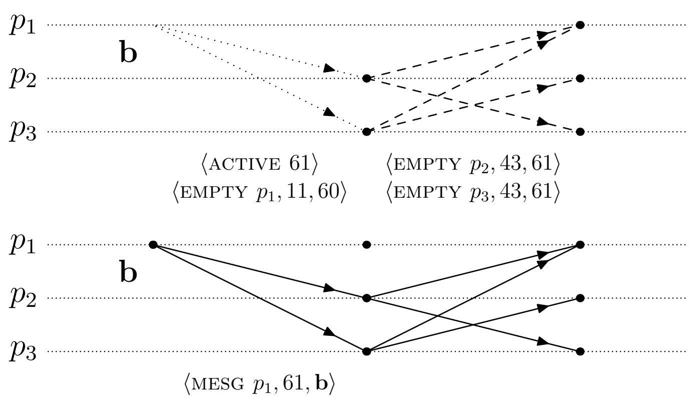


Figure 7: Typical message patterns involved in a single abcast.

6 Discussion

6.1 Message Complexity

Figure 7 shows messages related to abcasting a single message **b**. The upper diagram shows ordinary broadcast traffic related to ⟨ACTIVE⟩ and ⟨EMPTY⟩. The lower one presents a typical message pattern involved in Generic Broadcast of ⟨MESG $p_1, 61, \mathbf{b}$⟩. All messages in the upper diagram can therefore be piggybacked on the corresponding messages in the lower diagram. Thus, the message complexity of the Atomic Broadcast algorithm described here is the virtually the same as that of the underlying Generic Broadcast algorithm. This holds despite Atomic Broadcast achieving a two-step latency in *all* good runs, even in those with conflicting messages.

6.2 Inaccurate Clocks

In Fig. 4, assume p_3's clock is skewed and it timestamps **d** with 23 instead of 43. Message **c**, timestamped 32, will be delivered after **d**, timestamped with 23. If **d** is delivered in exactly two steps (time $123 = 43 + 2 \cdot 40$), then **c** will be delivered in two steps and $43 - 32 = 11$ units of time ($123 = 32 + 2 \cdot 40 + 11$). In general, the latency will be at most two communication steps plus the maximum clock difference Δ between two processes. By updating the clock whenever a process receives a message "from the future", one can ensure that Δ is at most one communication step. In the worst case, this reduces real-time clocks to scalar clocks [7], and ensures the latency of at most three steps in failure-free runs with inaccurate clocks.

6.3 Latency in Runs with Initial Failures

Consider a run in which all incorrect processes crash at the beginning and failure detectors do not make mistakes. In such runs, the current leader issues negative statements on behalf of the crashed processes. As opposed to negative *self*-statements, which use ordinary broadcast and are delivered in one step, the leader-issued ones use Generic Broadcast and take three steps to be delivered (or two if $n > 3f$). Therefore, the total latency in runs with initial/past failures grows from two to four steps (or three if $n > 3f$).

One communication step can be saved by eliminating line 11 from Fig. 3, so that every process (not only the current leader) can gbcast negative ⟨EMPTY⟩ statements on behalf of processes it suspects. In particular, any sender can now execute line 13 immediately after abcasting in lines 2–6, without waiting for the leader to receive its ⟨ACTIVE⟩ message in line 7. This saves one communication step and reduces the total latency to that of Generic Broadcast (two steps if $n > 3f$ and three otherwise).

6.4 Latency in Runs with General Failures

We have just seen that in runs with reliable failure detection, the total latency can be increased by at most one step. The picture changes considerably when failure detectors start making mistakes. First, if a crashed process is not (yet) suspected, no new messages can be delivered because the required ⟨EMPTY⟩ statements are not broadcast. Second, if a correct process is wrongly suspected, Generic Broadcast must deliver conflicting statements (from both the sender and the leader), which can significantly slow it down (Fig. 6c).

The above two problems cannot be solved at the same time: short failure detection timeouts motivated by the first increase the frequency of the other. To reduce the resultant high latency, the following technique can be used. When a process believes to be suspected, it should use the leader to abcast messages on its behalf rather than abcast them directly (a scheme similar to that in Sect. 4.3).

7 Lower Bounds

The Atomic Broadcast algorithm presented in this paper requires a majority of correct acceptors ($n > 2f$) and the $\Diamond S$ failure detector. These requirements are optimal [2], as is the latency of two communication steps [11]. Additional lower bounds hold for algorithms, such as this one or [13], which guarantee the latency lower than three communication steps in all good runs: Appendix Cproves that no Atomic Broadcast algorithm can guarantee latency lower than three steps in runs in which (i) processes do not have access to real time clocks, or (ii) external processes are allowed to abcast (the open-group model[4]), or (iii) a (non-leader) process fails.

Conditions (ii) and (iii) represent a trade-off between two-step and three-step algorithms. The latter usually allow external processes to broadcast, and guarantee good performance if at most f non-leader processes fail. On the other hand, two-step delivery requires synchronized real-time clocks, all processes correct, and no external senders. (External processes can still abcast by using the current leader as a relay, but this incurs an additional step.)

The algorithm shown here relies on $\Diamond S$ to ensure that faulty processes do not hamper progress. I suspect that Ω alone is insufficient to achieve a two-step latency in all good runs, however, this is still an open question.

8 Conclusion

The Atomic Broadcast algorithm presented in this paper uses local clocks to timestamp all abcast messages, and then delivers them in order of these timestamps. Processes broadcast both positive and negative statements ("message abcast" vs. "no messages abcast"). For fault-tolerance, the leader can communicate negative statements on behalf of processes it suspects to have crashed.

Negative self-statements do not conflict with anything, so they are announced using ordinary broadcast. Other statements are communicated using a new Generic Broadcast protocol, which ensures a two-step latency in conflict-and-failure-free runs, while requiring only $n > 2f$. Since no statements conflict in good runs, the Atomic Broadcast protocol described in this paper delivers all messages in two steps. Interestingly, this speed-up is achieved with practically no message overhead over the underlying Generic Broadcast. As opposed to [13], no network traffic is generated if no messages are abcast.

Although the presented algorithm is always correct (safe and live), it achieves the optimum two-step latency only in runs with synchronized clocks, no external processes, and no failures. These three conditions are required by any two-step protocol, which indicates an inherent trade-off between two- and three-step Atomic Broadcast implementations. It also poses an interesting question: how much power exactly do (possibly inaccurate) real-time clocks add to the asynchronous model?

Acknowledgements

I would like to thank the referees for many insightful comments and suggestions that allowed me to improve this paper.

References

[1] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Thrifty Generic Broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 268–282, Toledo, Spain, 2000.

[2] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[3] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[4] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[5] Paul Ezhilchelvan, Doug Palmer, and Michel Raynal. An optimal Atomic Broadcast protocol and an implementation framework. In *Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 32–41, January 2003.

[6] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Computer Science Department, May 1994.

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

[8] Achour Mostéfaoui and Michel Raynal. Low cost Consensus-based Atomic Broadcast. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing*, pages 45–52. IEEE Computer Society, 2000.

[9] NTP. Network Time Protocol, 2006. URL `http://www.ntp.org/`.

[10] Fernando Pedone and André Schiper. Optimistic Atomic Broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79–101, 2003.

[11] Fernando Pedone and André Schiper. On the inherent cost of Generic Broadcast. Technical Report IC/2004/46, Swiss Federal Institute of Technology (EPFL), May 2004.

[12] Fernando Pedone and André Schiper. Generic Broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 94–108, 1999.

[13] Pedro Vicente and Luís Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *Proceedings of 21st Symposium on Reliable Distributed Systems*, Osaka, Japan, 2002. IEEE Computer Society.

[14] Piotr Zieliński. Optimistic Generic Broadcast. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 369–383, Kraków, Poland, September 2005.

A Atomic Broadcast Proofs

Lemma A.1. *$time_i > t_i^{\max}$ at the beginning of every "block" (lines 2, 7, 10).*

Proof. Initially, $time_i > t_i^{\max} = 0$. Then, $t_i^{\max}$ is set only in lines 6 and 9 to values $\leq time_i$. The assertion holds because $time_i$ increases at the end of every "block" (after lines 6, 9, 13). □

Theorem A.2 (Uniform Integrity.). *For any message m, every process delivers m at most once, and only if m was previously broadcast.*

Proof. Process p_i atomically delivers m only if $(m, t) \in todeliver_i$. Since t is the unique timestamp assigned by $abcast(m)$, message m can be delivered at most once. Also, p_i must have delivered $\langle \text{MESG } p_j, t, m \rangle$ in line 17, which implies that p_j actually abcast m. Line 25 reabcasts m only if it cannot be delivered with its current timestamp. □

Lemma A.3. *Let p_i and p_j be processes. If p_i delivered m with timestamp t, and $t_j^{known} \geq t$, then p_j delivered m with timestamp t as well.*

Proof. The assumption implies that p_i received $\langle \textsc{mesg}\ p_k, t, m\rangle$ before any $\langle \textsc{empty}\ p_k, t_1, t_2\rangle$ with $t_1 \le t \le t_2$ for some p_k. Message $\langle \textsc{mesg}\ p_k, t, m\rangle$ must have been sent in line 5, using Generic Broadcast. Since statements $\langle \textsc{empty}\ p_k, t_1, t_2\rangle$ with $t_1 \le t \le t_2$ conflict with $\langle \textsc{mesg}\ p_k, t, m\rangle$, they must have been sent in line 13, also using Generic Broadcast.

As $t_j^{\text{known}} \ge t$, process p_j received at least one of the above statements, and since they have all been sent using Generic Broadcast, it also received $\langle \textsc{mesg}\ p_k, t, m\rangle$ first. Therefore, p_j added (m, t) to $\mathit{todeliver}_j$ and, since $t_j^{\text{known}} \ge t$, it delivered m with timestamp t. □

Lemma A.4. *If p_i and p_j both delivered m, they did so with the same timestamp.*

Proof. To obtain contradiction, assume that p_i delivered m with timestamp t_i, and p_j with timestamp $t_j \ge t_i$. Since $t_j^{\text{known}} \ge t_j \ge t_i$, Lemma A.3 implies that $t_j = t_i$. The case $t_j \le t_i$ is analogous. □

Theorem A.5 (Uniform Total Order.). *If some process p_i delivers message m' after message m, then every process p_j delivers m' only after it has delivered m.*

Proof. Assume p_i delivered m with timestamp t, and then m' with timestamp $t' > t$. If p_j delivered m', it must have done so with timestamp t' (Lemma A.4). Therefore $t_j^{\text{known}} \ge t' > t$, so Lemma A.3 implies that p_j delivered m with timestamp $t < t'$, that is, before m'. □

Lemma A.6. *If all correct processes receive $\langle \textsc{active}\ t\rangle$, then all correct processes will eventually have $t^{known} \ge t$.*

Proof. Receiving $\langle \textsc{active}\ t\rangle$ by a correct process p_i ensures that eventually $t_i^{\max} \ge t$ (line 9). Therefore, process p_i has announced all its abcast states up to time $t^{\max} \ge t$, so eventually $[1, t] \subseteq \mathit{known}_j[i]$ for all correct p_j and correct p_i.

Some correct process p_l will eventually: become the permanent leader (Ω), suspect ($\Diamond S$) all incorrect processes p_i, and have $t_i^{\max} \ge t$, possibly as a result of receiving $\langle \textsc{active}\ t\rangle$. When the last of these events happens, p_l will broadcast $\langle \textsc{empty}\ p_l, 1, \mathit{time}_l\rangle$ with $\mathit{time}_l \ge t_l^{\max} \ge t$ (line 13, Lemma A.1), so eventually $[1, t] \subseteq \mathit{known}_j[i]$ for all correct p_j and incorrect p_i.

The conclusions of the two last paragraphs imply the assertion. □

Theorem A.7 (Uniform Agreement.). *If a process delivers a message m, then all correct processes eventually deliver m.*

Proof. If a process atomically delivers a message m with timestamp t, then it must have delivered $\langle \textsc{mesg}\ p, t, m\rangle$ for some p in line 17. By Uniform Agreement of Generic Broadcast, each correct process p_j will eventually deliver $\langle \textsc{mesg}\ p, t, m\rangle$ and send $\langle \textsc{active}\ t\rangle$ to itself. Lemma A.6, ensures that all correct processes p_i will eventually have $t_i^{\text{known}} \ge t$. Lemma A.3 implies the conclusion. □

Lemma A.8. *Let p_i be a (correct) process that is eventually never suspected. Eventually, every message m abcast by p_i will be delivered by all correct processes p_j.*

Proof. If process p_i abcasts m finitely many times, then consider the last such abcast, at local time t. Since p_i is correct, all correct processes p_j will eventually deliver $\langle \text{MESG } p_i, t, m\rangle$ and add (m, t) to $\mathit{todeliver}_j$, because abcasting m again in line 25 contradicts the assumption that the previous abcast of m was the last one.

Consider the case when p_i abcasts m infinitely many times. Since p_i is eventually never suspected, there is a maximum t' for which $\langle \text{EMPTY } p_i, 1, t'\rangle$ in line 13 is broadcast. Therefore, one of the abcasts of m will happen at local time $t > t'$. As a result, $\langle \text{MESG } p_i, t, m\rangle$ will not conflict with any other statement, so all correct processes p_j will add (m, t) to $\mathit{todeliver}_j$.

We have proved that eventually $(m, t) \in \mathit{todeliver}_j$ for all correct p_j, whether m is abcast finitely many times or not. This implies that all correct processes p_j sent $\langle \text{ACTIVE } t\rangle$ to themselves, so by Lemma A.6 they will all eventually have $t_j^{\text{known}} \geq t$, which implies the assertion. □

With $\Diamond P$, all correct processes will eventually never be suspected, so Lemma A.8 implies:

Corollary A.9 (Validity). *If a correct process broadcasts a message m, then all correct processes will eventually deliver m.*

The modifications required for $\Diamond S$ were described in Sect. 4.3.

Theorem A.10 (Latency). *Let d be the maximum message delay between correct processes (possibly ∞). In good runs, a message m abcast at time t is delivered by all processes by time $t + 2d$.*

Proof. The assumption implies that each process p_i will receive $\langle \text{ACTIVE } t\rangle$ by time $t + d$, which will ensure $t_i^{\max} \geq t$. By definition of $t_i^{\max}$, process p_i announced its abcast state for each time $t' \leq t_i^{\max} \leq t$, by either: (i) broadcasting $\langle \text{MESG } p_i, t', m'\rangle$ in line 5, at time $t' \leq t$, using Generic Broadcast (2 steps), or (ii) broadcasting $\langle \text{EMPTY } p_i, t_1, t_2\rangle$ with $t_1 \leq t' \leq t_2$ in line 8, at time $t + d$ or before, using ordinary broadcast (1 step). In both cases, the appropriate statement will be delivered at all processes p_j by time $t + 2d$. As a result, $t' \in \mathit{known}_j[i]$ for all processes p_i and times $t' \leq t$, which ensures $t_j^{\text{known}} \geq t$.

The theorem assumption implies that each process p_j will deliver $\langle \text{MESG } p_i, t, m\rangle$ by time $t+2d$. Since good runs have no suspicions, no conflicting statements are broadcast, so $(m, t) \in \mathit{todeliver}_j$ by time $t + 2d$. Combining this with $t_j^{\text{known}} \geq t$ proves the assertion. □

B Generic Broadcast Proofs

Theorem B.1 (Validity). *If a correct process gbcasts a message m, then all correct processes will eventually deliver m.*

Proof. The assumption implies that all $n - f$ correct processes will receive $\langle \text{FIRST } m\rangle$, so all correct processes will receive $\langle \text{SECOND } * \ m\rangle$ from $n - f$ processes, so all correct processes will receive $\langle \text{THIRD } * \ m \ *\rangle$ from $n - f$ processes. If all correct processes receive only messages $\langle \text{THIRD good } m \ *\rangle$, then they will all deliver m in line 18. Otherwise, at least one correct process receives at least one message $\langle \text{THIRD bad } m \ *\rangle$. This process will then abcast $\langle \text{ATOMIC } m \ *\rangle$ and, by Validity of the underlying Atomic Broadcast, all correct processes will deliver m in line 22. □

Theorem B.2 (Uniform Agreement). *If a process delivers a message m, then all correct processes eventually deliver m.*

Proof. If a process delivers m in two steps (line 10), then it received $\langle$SECOND good $m\rangle$ from all processes. Therefore, all correct processes eventually receive $\langle$SECOND $*$ $m\rangle$ from $n - f$ processes, and all "*" are "good". All processes received $\langle$FIRST $m\rangle$ before any $\langle$FIRST $m'\rangle$ with $m' \in C(m)$, and $\langle$SECOND $*$ $m\rangle$ messages are FIFO-broadcast, so all processes receive $\langle$SECOND $*$ $m\rangle$ before any $\langle$SECOND $*$ $m'\rangle$ with $m' \in C(m)$. As a result, the condition in line 13 always holds for m. Thus, all correct processes broadcast $\langle$THIRD good m $\emptyset\rangle$, so all correct processes deliver m in line 18. No process abcasts $\langle$ATOMIC m $*\rangle$.

If a process delivers m in three steps (line 18), then it received $\langle$THIRD good m $*\rangle$ from $n - f$ processes, each of which received $\langle$SECOND good $m\rangle$ from $n - f$ processes, at least one of which is correct. Since $\langle$FIRST $m\rangle$ was reliably broadcast in line 3, all correct processes will receive $\langle$FIRST $m\rangle$, and the proof of Theorem B.1 implies the assertion.

If a process delivers m in line 21 or 22, then the Uniform Agreement property of Atomic Broadcast implies the assertion. □

Theorem B.3 (Uniform Integrity.). *For any message m, every process p delivers m at most once, and only if m was previously broadcast.*

Proof. The second part is straightforward as messages are only delivered if they have not been delivered before.

If p delivered m in two steps (line 10), then it received $\langle$SECOND good $m\rangle$ from all processes, each of which received $\langle$FIRST $m\rangle$ from the sender of m.

If p delivers m in three steps (line 18), then it received $\langle$THIRD $*$ m $*\rangle$ from some process, which received $\langle$SECOND $*$ $m\rangle$ from some process, which received $\langle$FIRST $m\rangle$ from the sender of m.

If p delivers m in line 22, then some process abcast $\langle$ATOMIC m $*\rangle$; the rest of the proof as above.

If p delivers m in line 21, then some process received $\langle$THIRD $*$ m' $\mathit{conflicts}_j\rangle$ with $m \in \mathit{conflicts}_j$. This means that $m \in \mathit{quick}_j$ at process p_j, which received $\langle$SECOND good $m\rangle$, which implies the assertion. □

Theorem B.4. *For any two conflicting messages m and m', it is impossible that one process p delivers m without having previously delivered m', and another process q delivers m' without having previously delivered m.*

Proof. If some process receives $n - f > n/2$ messages $\langle$SECOND good $m\rangle$, then a majority of processes received $\langle$FIRST $m\rangle$ before $\langle$FIRST $m'\rangle$. Similarly, if some process receives $n - f$ messages $\langle$SECOND good $m'\rangle$, then a majority of processes received $\langle$FIRST $m'\rangle$ before $\langle$FIRST $m\rangle$. Since majorities cannot be disjoint, assume, without loss of generality, that no process receives $n - f$ messages $\langle$SECOND good $m\rangle$. (In the other case, the symmetry of the assertion allows us to exchange $p \leftrightarrow q$ and $m \leftrightarrow m'$). Obviously, p cannot deliver m in lines 10 or 18. Since $m \in \mathit{quick}_i$ for no p_i, no message $\langle$THIRD $*$ m $\mathit{conflicts}_i\rangle$ with $m \in \mathit{conflicts}_i$ is ever sent, so p cannot deliver m in line 21 either. The only possibility left is p delivering m in line 22 after atomically delivering $\langle$ATOMIC m $*\rangle$ in line 20.

In which line can q deliver m'? Since p delivers m without having previously delivered m', no $\langle \textsc{atomic}\ m'\ *\rangle$ or $\langle \textsc{atomic}\ *\ \mathit{conflicts}\rangle$ with $m' \in \mathit{conflicts}$ is delivered by p before $\langle \textsc{atomic}\ m\ *\rangle$. The Total Order of Atomic Broadcast implies that if q delivers m' in line 21 or 22, it must have delivered m before.

Since p delivers m in line 22 without having previously delivered m', some process must have abcast $\langle \textsc{atomic}\ m\ \mathit{conflicts}\rangle$ with $m' \notin \mathit{conflicts}$. This means that a majority of processes p_i broadcast $\langle \textsc{third}\ *\ m\ \mathit{conflicts}_i\rangle$ with $m' \notin \mathit{conflicts}_i$, so they must have added m to seen_i^2 in line 12 before adding m'.

If q delivers m' in line 18, then it received $\langle \textsc{third}\ \text{good}\ m'\ *\rangle$ from a majority of processes p_i, which must have added m' to seen_i^2 before adding m. The previous paragraph showed that a majority of processes p_i added m and m' to seen_i^2 in the opposite order. This contradiction proves that q could not deliver m' in line 18.

Finally, if q delivered m' in line 10, then all processes received $\langle \textsc{first}\ m'\rangle$ before $\langle \textsc{first}\ m\rangle$. By the FIFO property, no process p_i added m to seen_i^2 before m', however, we proved that a majority of processes did so. This final contradiction shows that q could not deliver m' without delivering m before, which proves the assertion. □

Corollary B.5 (Uniform Partial Order.). *If some process p_i delivers message m' after message m conflicting with m', then every process p_j delivers m' only after it has delivered m.*

C Lower Bounds

This section proves several impossibility results for Atomic Broadcast protocols that tolerate at least one faulty process ($f > 0$).

C.1 Latency below three steps requires real-time clocks

Theorem C.1. *Only Atomic Broadcast algorithms that use real-time clocks can guarantee a latency of less than three communication steps in all good runs.*

Proof. To obtain a contradiction, assume the existence of an Atomic Broadcast algorithm that does not use real-time clocks but in good runs delivers all messages within $K < 3$ communication steps. I will show that such an assumption leads to a contradiction.

Consider a family of good runs $r(k)$ for $k = 0, 1, \ldots, n$, in which processes p_1 and p_2 abcast two messages m_1 and m_2, respectively, at time 0, and no other messages are abcast. All processes are correct and almost all messages have the latency of d. The only exceptions are some messages sent at time 0: those from process p_1 to processes $p_1, \ldots, p_k$, and those from p_2 to $p_{k+1}, \ldots, p_n$. These messages have the latency of $d - \varepsilon$, for some small $\varepsilon > 0$ which will be defined later. All other messages have a latency of d.

I will first prove that, for any $k = 1, \ldots, n$, runs $r(k)$ and $r(k-1)$ deliver messages m_1 and m_2 in the same order. For each $i \in \{k-1, k\}$, consider the run $r_k(i)$, which is identical to $r(i)$, except that all messages sent by process p_k to other processes at time $d - \varepsilon$ or later are lost. Runs $r(i)$ and $r_k(i)$ are identical until time $d - \varepsilon$. Since all messages sent after time 0 have latencies d, runs $r(i)$ and $r_k(i)$ are indistinguishable to processes other than p_k until time $2d - \varepsilon$, and to p_k itself until $3d - \varepsilon$. Since $K < 3$, we

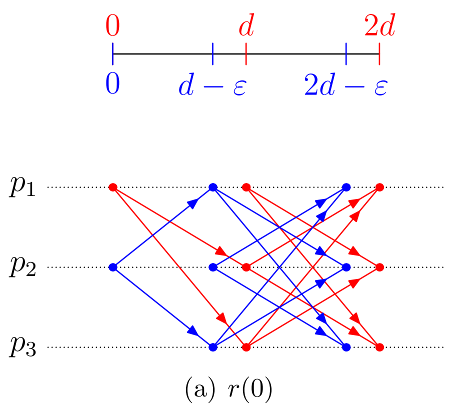


Figure 8: Runs $r(0)$ and r' used in the proof of Theorem C.1.

have $3d - \varepsilon > Kd$ for sufficiently small ε. This means that process p_k delivers the same message first in both runs $r(i)$ and $r_k(i)$. Uniform Total Order and Uniform Agreement imply that all correct processes deliver the same message first in $r(i)$ and $r_k(i)$.

To show that the runs $r(k)$ and $r(k-1)$ deliver the same message first, it is then sufficient to show the same for runs $r_k(k)$ and $r_k(k-1)$. Runs $r_k(k)$ and $r_k(k-1)$ differ only in the delays of messages sent by processes p_1 and p_2 to process p_k at time 0. However, these messages arrive at p_k at time $d-\varepsilon$ or later, and from that time all messages from p_k to other processes are lost. Therefore, these two runs are indistinguishable to any correct process $p \neq p_k$, which delivers the same message first in both of them.

We have shown that, for any $k = 1, \ldots, n$, runs $r(k)$ and $r(k-1)$ deliver messages m_1 and m_2 in the same order. Simple induction on k shows that the same is true for runs $r(0)$ and $r(n)$. Without loss of generality, assume m_1 is delivered first in both runs, and focus on run $r(0)$. (The other case, in which m_2 is delivered first, is analogous and requires considering run $r(n)$.) In run $r(0)$, shown in Fig. 8, all message latencies are d, except for those sent by process p_2 at time 0; these have the latency of $d - \varepsilon$.

Consider a good run r' which is identical to $r(0)$ except that process p_1 abcasts m_1 at time $d - 2\varepsilon$ instead of at time 0. Figure 8 shows that runs r' and $r(0)$ are causally identical, so processes without real-time clocks cannot distinguish them. As a result, the same message (m_1) is delivered first in both of them.

In run r', process p_1 cannot deliver m_1 before getting feedback from other processes, which takes two communication steps (until time $3d - 2\varepsilon$). Since message m_2, abcast at time 0, is delivered after m_1, process p_1 cannot delivered it before time $3d-2\varepsilon$ either. Since $K < 3$, this is bigger than Kd for sufficiently small ε, which contradicts the assumption that, in good runs, all messages are delivered in $K < 3$ steps. □

C.2 Dealing with faulty processes requires three steps

Definition C.2. *A run is* timely *if failure detectors do not suspect any correct processes.*

Theorem C.3. *Consider two (possibly external) processes q_1 and q_2 No Atomic Broadcast algorithm can guarantee a latency of less than three communication steps in all timely runs with all processes correct except for possibly one of q_1 and q_2.*

By taking q_1 and q_2 as external processes, we obtain

Theorem C.4. *No Atomic Broadcast algorithm that allows external processes to abcast messages can guarantee a latency of less than three communication step in all good runs.*

On the other hand, taking q_1 and q_2 to be two main processes other than the leader, we get

Theorem C.5. *No Atomic Broadcast algorithm can guarantee a latency of less than three communication step in all timely runs with at most one non-leader process being faulty.*

To prove Theorem C.3, assume that such an Atomic Broadcast algorithm does exist. We will be considering runs in which process q_1 abcasts message m_1, and process q_2 message m_2, both at time 0. First, consider a run r, in which all processes are correct and all messages have latencies d. Without loss of generality, we can assume that message m_1 is delivered first in this run. Otherwise, we can simply exchange the roles of processes q_1 and q_2.

Consider a family of runs $r(k)$ with $k = 0, \ldots, n$. Run $r(k)$ is identical to r, except that process q_1 is faulty and crashes at some time in the future, say $3d$. All messages sent by process q_1 to processes $p_1, \ldots, p_k$ have the latency $3d$ instead of d; the latencies of messages from q_1 to other processes remain d. Process q_2 is correct. All messages between correct processes have a latency of d, so all correct processes deliver message m_2 before time $3d$.

The only difference between run r and $r(0)$ is the correctness of process q_1; these runs are indistinguishable to any process before time $3d$, so $r(0)$ delivers message m_1 first. I will later show that, for any k, runs $r(k-1)$ and $r(k)$ deliver the same message first. By simple induction, run $r(n)$ delivers message m_1 before m_2. We have previously seen that, in run $r(n)$, all correct processes deliver m_2 before time $3d$, so m_1 must also be delivered before that time. However, in run $r(n)$, no process other than q_1 knows about m_1 before time $3d$, so it is impossible for all correct processes to deliver m_1 before that time. This contradiction proves the assertion.

Runs $r(k-1)$ and $r(k)$ deliver the same message first

For each $i \in \{k-1, k\}$ consider a run $r_k(i)$ which is identical to $r(i)$, except that process p_k is faulty and crashes at time $3d$, and process q_1 is correct (unless $q_1 = p_k$). All messages from p_k to other processes sent at time d or later are lost. Runs $r(i)$ and $r_k(i)$ are identical until time d, so they are indistinguishable to processes other than p_k until time $2d$. For process p_k, these two runs seem the same until time $3d$.

We have already shown that, in run $r(i)$, all correct processes, in particular process p_k, deliver at least one message (m_2) before time $3d$. Since p_k cannot distinguish runs $r(i)$ and $r_k(i)$ before that time, it delivers the same message first in both runs. Uniform Total Order and Uniform Agreement imply that all correct processes deliver the same message first in runs $r(i)$ and $r_k(i)$.

To show that runs $r(k-1)$ and $r(k)$ deliver the same message first, it is now sufficient to show the same for runs $r_k(k-1)$ and $r_k(k)$. Before time $3d$, these runs differ only in the latency of messages from process q_1 to process p_k. These two runs are indistinguishable to p_k until time d but, from that time on, all messages from p_k to other processes are lost. As a result, other processes cannot distinguish runs $r_k(k-1)$ and $r_k(k)$ before time $3d$, so they must deliver the same message first in both runs. This way, we have proved that runs $r(k-1)$ and $r(k)$ deliver the same message first.