NOTES ON ATOMIC BROADCAST

LISA HIGHAM AND JOLANTA WARPECHOWSKA-GRUCA

Abstract. Atomic broadcast [3] is a powerful communication primitive, which is applied in a natural way in sequentially consistent implementations of various data structures (see [1]). Unfortunately the atomic broadcast algorithm as given in [1] is incorrect. Here some ways of correcting it are analyzed under assumptions of blocking versus nonblocking communication.

Two atomic broadcast algorithms are proposed: one correct if it is used in a blocking manner, the other correct unconditionally. The latter algorithm, when used in the sequentially consistent implementations of data structures proposed in [1], has the same time complexity and a reduced message complexity compared to that claimed in the original application[1].

Keywords: Sequential consistency, Atomic broadcast, Blocking execution.

1. Introduction

The overhead of ensuring linearizability versus sequential consistency of shared objects in a message passing system under various realistic timing assumptions is well studied in [1]. This note points out that the algorithms for sequential consistency are flawed for the model assumed and corrects the algorithms. The errors are subtle; one hinges on issues involving blocking versus nonblocking communication. Specifically, in the original paper, sequentially consistent implementations of several data structures (read/write registers, queues, stacks) are designed using a communication primitive called *atomic broadcast*. The operations on the data-structures are assumed to be blocking. However, if atomic broadcast is also assumed to be blocking, then the claimed time complexities of the sequentially consistent implementations do not hold. On the other hand, if atomic broadcast is assumed to be nonblocking, then the broadcast algorithm provided is incorrect. Similar problems appear in the asynchronous implementation of $FIFO$ queues in [2].

In section 2 the assumptions about the computation model are stated, an atomic broadcast primitive is defined, and the architecture of a memory consistency system that uses atomic broadcast is clarified.

[1]These notes should be read as a supplement to the original paper [1].

In section 3, it is shown that the original atomic broadcast algorithm [1] can deadlock. Several plausible repairs are examined and shown to be inadequate. A correction is proposed under the assumption that atomic broadcast is blocking. Next, it is shown that if atomic broadcast is blocking, the sequentially consistent implementations of data structures do not have the time complexity previously claimed.

In section 4 it is shown that if atomic broadcast is not required to be blocking, then even the repaired algorithm of section 3 can deadlock. A new nonblocking broadcast algorithm is developed and proved correct. We then show that the time complexity originally claimed for the sequentially consistent implementations holds if this revised nonblocking broadcast is used, and that the revised broadcast reduces the message complexity.

2. Definitions

To establish lower bounds as well as upper bounds for both linearizable and sequentially consistent systems under various timing constraints, a general and formal description of the system is required and has been provided by Attiya and Welch [1]. Some simplification is possible for this note since we are establishing only upper bounds for sequentially consistent systems for one timing option.

An operation is an ordered pair of *invocation* and *response* events. A *shared object* is specified by a set of sequences of operations. A sequence of operations τ is *legal* for a collection C of shared objects if, for each object $o \in C$, the subsequence of τ consisting of operations on o is in the specification of o. A system consists of a collection of *nodes* connected through a communication network. On each node there is an application process, and a *memory consistency system* (MCS) process. Each application process can access a shared object by issuing an invocation to its corresponding MCS process and receiving a response to this operation from its MCS process. Each MCS process is a transition automaton that is driven by input events: invocation (from the application process) and message receive (from the network), and produces output events: message send (to the network) and response (to the application process), as shown in figure 1. For each pair of nodes p and q, messages sent from p to q are guaranteed to arrive in the order sent, and each message arrives at its destination after a bounded delay. Delays are bounded within an interval $[d-u, d]$, for some constants $d \geq 0$ and $0 \leq u \leq d$. It is assumed that for every process at most one invocation is pending at a time.

An *execution* of a process is a sequence of pairs: (event, real time) non-decreasing with respect to the second component. For an application process an event is an operation invocation or an operation response. For an MCS process an event is either an operation invocations, an operation response, a message send or a message receive. The subsequence of an MCS process execution consisting only of those pairs with an invocation or a response event is identical to the application process execution at the same node. An execution of a system is a set of MCS processes executions. An event of process p sending message m to process q and an event of process q receiving message m from process p are called matching message send and message receive events. An *admissible* execution is one in which there is a one-to-one correspondence between matching messages sent and message receive events, and the difference between the real time of a message receive event and a matching message send event is within $[d-u, d]$.

Consider an execution of any application process. Since at most one invocation is pending at a time, every response matches the immediately preceding invocation. Therefore, the execution of an application process can be viewed as a sequence of operations tagged with real times. The execution of the system is *sequentially consistent* if there exists a legal total order on the operations of all the processes that extends the order given by each process's sequence of operations.

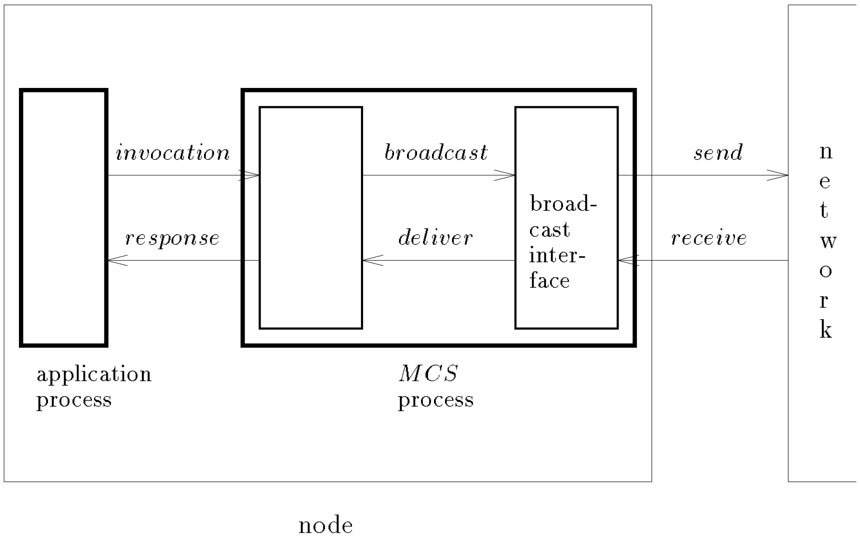

FIGURE 1. Memory consistency system using atomic broadcast

An *atomic broadcast* is a communication primitive that receives a sequence of *broadcast* instructions as input from each process and produces a sequence of *deliver* instructions to each process. Each broadcast instruction contains a *broadcast message*, and the atomic broadcast must satisfy:

(1) Every broadcast message is delivered at every process,
(2) All broadcast messages are delivered in the same order at all processes,

(3) Two broadcast messages by the same process are delivered in the same order as they were broadcast.

Atomic broadcast is called *blocking* if, for every process p, at most one broadcast is pending at a time. Otherwise, it is called *nonblocking.* The *time complexity* of an atomic broadcast algorithm is the maximum time it takes to deliver a broadcast message to all processes.

Our goal is to find a distributed implementation of the atomic broadcast primitive such that when used in the construction of an MCS process (as shown in figure 1) the resulting system is sequentially consistent.

3. Blocking atomic broadcast

The atomic broadcast algorithm proposed in [1] is shown in figure 2 (where *ABC-send* is the broadcast instruction and *ABC-receive* is the deliver instruction).

```
ABC-send_p(m):
    send (ts_p[p], m) to all processes
    ts_p[p] ← ts_p[p] + 1
receive (t, m) from q:
    add (m, t, q) to pending_p
    if t + 1 > ts_p[p] then
        ts_p[p] ← t + 1
        send timestamp (ts_p[p]) to all processes
receive timestamp (t) from q:
    ts_p[q] ← t
    repeat
        let E be element with smallest (ts, id) pair in pending_p
        if for some q, ts_p[q] ≤ E.ts then exit fi
        deliver E.m {this is ABC-receive_p}
        remove E from pending_p
    end repeat
```

FIGURE 2. Algorithm 1

Claim 3.1. *There exists an execution of Algorithm 1 in which no broadcast is delivered.*

Proof. Consider an execution, where two processes p and q, both having timestamp 0, broadcast messages at the same time. Both broadcasts carry timestamp 0, and the timestamps of both processes will increase to 1. Upon receiving any of the messages, neither of the processes will send a timestamp message, since $0+1 \not> 1$. Since delivery happens only upon receiving a timestamp message, no broadcast is delivered.

If atomic broadcast is blocking, then no process can execute another *ABC-send* because its previous broadcast is not delivered.

If atomic broadcast is nonblocking, and the processes continue to send messages in lock-step, messages in transit will carry timestamps strictly less than any of the timestamps of the processes. No timestamp message will be sent, and since broadcasts are delivered only upon receiving a timestamp message, no broadcast will be delivered. □

Several simple changes to this algorithm may suggest themselves as possible repairs. Under the assumption that atomic broadcast is blocking, the problem of deadlock can be rectified by either (1) changing the test in line 4 to "if $t+1 \geq ts_p[p]$ then" or (2) swapping the order of the two instructions of *ABC-send*: a process first increases its timestamp, and then sends a message with the increased timestamp to all other processes. A stronger measure than (1), namely unconditional sending of a timestamp upon receiving a broadcast message, rectifies the problem of deadlock even when atomic broadcast is nonblocking. Unfortunately these ways to remove deadlock expose other errors. Observe that, upon receiving a timestamp message from q, process p unconditionally changes $ts_p[q]$ to the value of the timestamp message. If $p = q$, and if p has received messages with higher timestamp between sending and receiving its own timestamp message, then this can cause the timestamp of process p to decrease. This, in turn, can jeopardize delivering the broadcasts in timestamp order, and as a consequence violates the condition that broadcasts are delivered to all processes in the same order. Examples of erroneous executions illustrating this phenomenon are presented in appendix A.

In order to design a correct broadcast algorithm, it is illuminating to observe where the original proof of correctness of Algorithm 1 (figure 2) fails. The correctness is claimed to follow from the following three lemmas where σ denotes any admissible execution.

Lemma 3.2. *Let p be any process. Then every message broadcast by p in σ is given a unique timestamp in increasing order.*

Lemma 3.3. *The timestamps assigned to messages in σ, together with process identifiers form a total order.*

Definition: The total order of lemma 3.3 is called the *timestamp order*.

Lemma 3.4. *Let p be any process. Then all broadcast messages are delivered to p in σ in timestamp order.*

First, lemma 3.2 is incorrect if broadcast is blocking, and not obvious even in blocking case. (The code $\mathrm{ts}_p[q] \leftarrow t$ of line 7 can cause timestamps to decrease in the case that the message received is from p itself.) Next, in the argument of lemma 3.4, there is some reasoning about sending of timestamps that concludes "p must have received a timestamp message from q_1" which is incorrect when $p = q_1$ (see appendix A: figure 7 and figure 8). Finally, these lemmas (or more accurately, their proofs) fail to establish freedom from deadlock or livelock. Although the statement of lemma 3.4 claims all messages are delivered, the argument, even when corrected, would still only imply that those messages that are delivered, are delivered in timestamp order. It is conceivable that some messages are not delivered or not even broadcast (as was demonstrated in claim 3.1).

Provided that atomic broadcast is blocking, Algorithm 1 can be corrected by (1) repairing the deadlock and (2) distinguishing between the timestamp that is updated in lines 2 and 5 (used for sending) and the timestamp that is updated in line 7 upon receipt of the process's own timestamp message. This revision, called Blocking-Broadcast, is show in figure 3. The

$broadcast_p(m)$:
    send (my_ts_p, m) to all
    $my\_ts_p \leftarrow my\_ts_p + 1$

receive (t, m) from q:
    add (m, t, q) to $pending_p$
    if $t + 1 \geq my\_ts_p$ then
        $my\_ts_p \leftarrow t + 1$
        send timestamp (my_ts_p) to all

receive timestamp (t) from q:
    $ts_p[q] \leftarrow t$
    repeat
        let $E = (m, t, r)$ be the minimum element in $pending_p$ by timestamp order
        if for some q, $ts_p[q] \leq E.t$ then exit fi
        deliver $E.m$
        remove E from $pending_p$
    end repeat

FIGURE 3. Algorithm Blocking-Broadcast

proof that Blocking-Broadcast is a correct implementation of blocking atomic broadcast and has time complexity $2d$ under our timing assumptions is inspired by arguments in the original paper [1]. Fix an execution σ of Blocking-Broadcast. Observe that when a process sends a message, whether a broadcast message or a timestamp, it sends that message to every process. Call such a set of messages a *communication.*

Lemma 3.5. *For every process p, every communication of a broadcast message sent by p in σ has a unique timestamp and these timestamps are strictly increasing. The timestamps of all communications (both broadcast and timestamp) sent by process p are non-decreasing.*

Proof. Process p's broadcast communication is immediately followed by an increase in p's timestamp my_ts_p. Timestamp communication contains process p's timestamp my_ts_p. The lemma follows immediately from the observation that my_ts_p never decreases. □

Lemma 3.6. *The broadcast communications are totally ordered by timestamp order.*

Proof. This follows from lemma 3.5, the fact that one copy of each broadcast message is sent to each process, and the fact that processes have unique identifiers. □

Lemma 3.7. *For every process p, all broadcast messages that are delivered to p in σ, are delivered in timestamp order.*

Proof. Suppose for the purpose of contradiction, that message m_1 with label (t_1, q_1) is delivered by p after message m_2 with label (t_2, q_2) is delivered by p, and that $(t_1, q_1) < (t_2, q_2)$. It follows that when m_2 is delivered by p: (1) $ts_p[q_1] > t_2$ and (2) m_1 is not in the list pending$_p$ and hence is received after m_2 is delivered. Since entries of ts_p vector change only upon receiving a timestamp message, p must have received from q_1 a timestamp message $t > t_2 \geq t_1$. Since $t > t_1$, by lemma 3.5, process q_1 sent message (t_1, m_1) before sending timestamp message t. By the $FIFO$ property of the communication links, process p received message (t_1, m_1) from q_1 before receiving timestamp message t from q_1, which contradicts the fact that m_1 is not in the list pending$_p$ when m_2 is delivered. □

The three previous lemmae establish that, assuming that each broadcast message is delivered by each process, they will be delivered in the same order by every process and in an order that extends the process order. The following lemma, which establishes the time complexity of algorithm Blocking-Broadcast, immediately implies that each broadcast message is delivered by each process. It is this lemma that requires broadcast to be blocking.

Lemma 3.8. *In Blocking-Broadcast every broadcast message is delivered within time $2d$.*

Proof. Suppose process p broadcasts a message m with timestamp x at time t. Then process p sends message (x, m) to all processes, and increases its timestamp to $x + 1$. By time $t + d$, message (x, m) from p is received by every process, and is added to the list of pending messages of each process. For every process r, let $t_r \leq t + d$ denote the time when r receives message (x, m) from p.

Consider any process q that has $my\_ts_q \leq x + 1$ at time t_q. Then q will send a timestamp with value $x + 1$ at time t_q, which will be received by all processes by time $t_q + d \leq t + 2d$.

Consider any process s that has $my\_ts_s = x' > x + 1$ at time t_s. Let $t' < t_s$, be the time when s changed my_ts_s from a value less than or equal to x to a value strictly greater than x. If that increase was caused by receipt of a message then s must have sent a timestamp grater than x at time t'. If that increase was caused by a broadcast then my_ts_s must have been incremented from x to $x + 1$. In this case there must have been another increase between time t' and t_s, from $x + 1$ to x'. Since broadcast is blocking, this last increase must have been due to receipt of a message, in which case s sent a timestamp with value x', which will be received by all processes by time $t + 2d$.

Therefore, each process r will have received a timestamp with value strictly greater than x from every process by time $t + 2d$. At least one of these (in particular, the one from p), will be received after time t_r. The last such timestamp received will cause r to deliver (x, m). □

Unfortunately, as the next claim shows, if the atomic broadcast primitive used in the sequentially consistent fast-write algorithm [1] (shown in figure 4) is blocking, then the implementation does not achieve time complexity $2d$ as was claimed.

```
Read_p(X):
    if num = 0 then generate Ret_p(X, v),
        where v is the value of p's copy of X fi
Write_p(X, v):
    num ← num + 1
    ABC-send(X, v)
    generate Ack_p(X)
ABC-receive(X, v) from q:
    set local copy of X to v
    if p = q then
        num ← num − 1
        if num = 0 then generate Ret_p(X, v),
            where v is the value of p's copy of X fi
    fi
```

FIGURE 4. Fast-Write Algorithm

Claim 3.9. *If the Fast-Write Algorithm uses blocking atomic broadcast and is sequentially consistent, then the time of a Read operation is unbounded.*

Proof. Let $n \in \mathbb{N}$ be arbitrary. We construct an execution, where *Read* returns after time greater than $2nd$. Suppose the application process executes the sequence of operations: $Write(X, 0)$, $Write(X, 1), \ldots, Write(X, n)$, $Read(X)$. Suppose, the corresponding MCS process receives these invocations one after another at times $t, (t+\epsilon), \ldots, (t+n\epsilon), (t+(n+1)\epsilon)$, where $\epsilon < d/(n+1)$. This is possible, since $Write$ is acknowledged immediately. However, atomic broadcast is blocking. Therefore, the MCS process executes the broadcast for the first write, and is able to execute broadcast for the next write only after delivering the previous one. Suppose all message delays are exactly d. Since sequential consistency has to satisfy the process order, the *Read* operation that started at time $t + (n+1)\epsilon$ can return only after delivering the last $Write$, which can be as late as $t + 2(n+1)d < t + d$. □

The same problem as illustrated for read/write registers in claim 3.9 arises for the sequentially consistent queue and stack algorithms in [1].

4. Nonblocking atomic broadcast

The next claim shows that Blocking-Broadcast, presented in figure 3, is incorrect for nonblocking atomic broadcast.

Claim 4.1. *If atomic broadcast is nonblocking, then in algorithm Blocking-Broadcast it is possible that no broadcast is delivered.*

Proof. In the following execution with one process p no timestamp message is sent, and therefore no broadcast is delivered. Process p starts at time 0 by executing $broadcast_p(m_0)$: it sends message $(0, m_0)$ and increases its timestamp to 1. Just before receiving this message, p broadcasts m_1 sending message $(1, m_1)$ and increasing timestamp to 2. Thus when it receives message $(0, m_0)$, its timestamp is already 2, so no timestamp message is sent.

Extend this to an infinite history. For any natural number k, just before receiving its own message (k, m_k), process p broadcasts message m_{k+1} increasing its timestamp to $(k + 2)$. When p receives message (k, m_k), it does not send a timestamp.

Extend this idea to a system with an arbitrary number of processes, each with an identical history as described above and each operating in lockstep. When messages with timestamp k arrive, all processes have timestamps equal to $(k + 2)$. Hence, no timestamp message is sent, and no message delivered. □

To find a correct nonblocking broadcast algorithm, we first observe that Algorithm 1 is a nearly correct nonblocking atomic broadcast for a stronger model than the one we actually have. Specifically, suppose the model is one in which for all processes p, all messages sent from p to p are received instantaneously and are immediately processed by p; that is, they go to the head of the input queue of messages from the network for p. All messages from p to $q \neq p$ meet the original constraints on delay. Call this the *strong* timing assumption, as opposed to the *weak* timing assumption that we actually have. The erroneous executions of claim 4.1 and figures 7 and 8 in appendix A are not possible given the strong timing assumption. As already pointed out, it is easy to remove the deadlock illustrated by claim 3.1. Furthermore we can simulate the behaviour of an algorithm under the strong timing assumption, when we have only the weak timing assumption by never using the network to send a message from a process to itself.

This strategy is implemented in the remainder of this section as follows: Define Algorithm 2 to be the same as Algorithm 1 (figure 2 on page 4) except line 4 is changed to: "if $t+1 \geq ts_p[p]$ then". We first prove Algorithm 2 is a correct and efficient nonblocking atomic broadcast algorithm under the strong timing assumption. Then we develop an algorithm for a model that guarantees only the weak timing assumption, that exactly simulates the behaviour of Algorithm 2 under the strong timing assumption. Finally we further transform the algorithm for the weak model to one that has the equivalent pattern of broadcast and delivery and that uses fewer messages.

The proof of correctness of Algorithm 2 under the strong timing assumption again has structure similar to that of the original argument and to that used earlier in this paper. Let σ be an admissible execution of Algorithm 2 under the strong timing assumption.

Lemma 4.2. *Given the strong timing assumption, for every process p, every communication of a broadcast message sent by p in σ has a unique timestamp and these timestamps are strictly increasing. The timestamps of all communications sent by p are non-decreasing.*

Proof. Process p's broadcast communication is immediately followed by an increase in p's timestamp. Thus as long as p's timestamp is never decreased, we are done. The only other adjustments to p's timestamp are in lines 5 and 7. Line 5 can only change the timestamp to the same value or an increased value. In the strong model p will always follow its broadcast message (x, m) with a timestamp message $(x + 1)$ and will instantly receive that timestamp message $(x + 1)$ and thus make no change to its timestamp in line 7 of the algorithm. □

Lemma 4.3. *Given the strong timing assumption, for every process p, the timestamp order imposes a total order on the broadcast messages received by p.*

Proof. This follows immediately from lemma 4.2, the fact that one copy of each message is sent to each process, and the fact that processes have unique identifiers. □

Lemma 4.4. *Given the strong timing assumption, for every process p, every message that is delivered by p in σ is delivered in timestamp order.*

Proof. Suppose for the purpose of contradiction, that message m_1 with label (t_1, q_1) is delivered by p after message m_2 with label (t_2, q_2) is delivered by p, and that $(t_1, q_1) < (t_2, q_2)$. It follows that when m_2 is delivered: (1) $ts_p[q_1] > t_2$ and (2) m_1 is not in the list pending$_p$ and hence is received after m_2 is delivered. There are two cases.

Case 1: $q_1 \neq p$. Then p received a timestamp $t \geq t_2 + 1$ from q_1 before m_2 is delivered. But $(t_1, q_1) < (t_2, q_2)$ implies $t_1 \leq t_2$ and hence $t > t_1$. Thus, by lemma 4.2, q_1 sent message m_1 to p before it sent timestamp t to p. Since the communication channels are FIFO, this contradicts that m_1 is received by p after m_2 is delivered.

Case 2: $q_1 = p$. Say $ts_p[p] = t' > t_2$ at the time of delivery of m_2. Because of the strong timing assumption, line 7 has no effect on the value of $ts_p[p]$, so $ts_p[p]$ must have been first set to t' at line 2 or line 5. If it was set at line 5, then it was accompanied by sending and instantaneous receipt of a timestamp $t' > t_1$ so m_1 must have been previously sent. Then, by the strong timing assumption, m_1 must have been previously received, contradicting the assumption that it is received after m_2. If it was set at line 2, then it was immediately preceded by sending of a message with timestamp $t' - 1 \geq t_1$. So again, by the strong timing assumption and lemma 4.2, this contradicts the assumption that m_1 is received after m_2. □

By the three previous lemmas, assuming that each broadcast message is delivered to each process, it follows that they will be delivered in the same order at each process and in an order that extends the process order. The following lemma, which establishes the time complexity of Algorithm 2 assuming the strong model, has as an immediate corollary that each broadcast message is delivered at each process.

Lemma 4.5. *Given the strong timing assumption, in Algorithm 2 every broadcast message is delivered within time $2d$.*

Proof. Suppose process p executes $ABC\text{-}send_p(m)$ at time t. So it communicates message (x, m), increases its timestamp to $x + 1$ and then communicates timestamp $x + 1$. By time $t + d$ the message (x, m) is received by every process, and added to the list of pending messages of every process, and the timestamp $(x + 1)$ is received by every process.

For any process r, let $t_r \leq t + d$ denote the time that process r receives the timestamp $x + 1$ from p. Suppose there is a process s that cannot deliver message (x, m) by time $t_s \leq t + d$. Then there exists a process $q \neq p$ such that at time t_s process s has $ts_s[q] \leq x$. Let x' be process q's timestamp $ts_q[q]$ just before time t_q. If $x' \leq x + 1$, then at time $t_q \leq t + d$, when q receives (x, m) from p, it will communicate a timestamp $x + 1$, which will be received by

s by time $t_q + d \leq t + 2d$. If $x' > x + 1$, consider the moment prior to t_q when $ts_q[q]$ was increased to x'. Because of the strong timing assumption, this increment could not have been in line 7. If this increment was in line 2 or line 5 then q communicated a timestamp $x' > x + 1$ which will be received by s by time $t_q + d \leq t + 2d$.

Therefore, for every process q for which at time t_s the value $ts_s[q] \leq x$, process s receives a timestamp from q with value at least $x + 1$ by time $t + 2d$. The last such timestamp received will cause s to deliver (x, m). $\square$

```
broadcast_p(m):
      send (ts_p[p], m) to all q ≠ p
      add (m, ts_p[p], p) to pending_p
      ts_p[p] ← ts_p[p] + 1
      send timestamp (ts_p[p]) to all q ≠ p
      delivery progress
receive (t, m) from q ≠ p:
      add (m, t, q) to pending_p
      if t + 1 ≥ ts_p[p] then
            ts_p[p] ← t + 1
            send timestamp (ts_p[p]) to all q ≠ p
            delivery progress
      fi
receive timestamp (t) from q ≠ p:
      ts_p[q] ← t
      delivery progress
delivery progress:
      repeat
            let E = (m, t, r) be the minimum element in pending_p by timestamp order
            if for some q, ts_p[q] ≤ E.t then exit fi
            deliver E.m
            remove E from pending_p
      end repeat
```

FIGURE 5. Algorithm Nonblocking-Broadcast-1

Algorithm Nonblocking-Broadcast-1 in figure 5 is constructed from Algorithm 2, by simulating the delivery of all messages from a process to itself by bypassing the network and proceeding exactly as if each such message is immediately received. The conversion is transparent and thus the proof of correctness and complexity of Nonblocking-Broadcast-1 under the weak timing assumption is an immediate corollary of the same result for Algorithm 2 under the strong timing assumption.

$broadcast_p(m)$:
  add $(m, ts_p[p], p)$ to $pending_p$
  $ts_p[p] \leftarrow ts_p[p] + 1$
  send $(ts_p[p], m)$ to all $q \neq p$
  delivery progress

receive (t, m) from $q \neq p$:
  if $m \neq \perp$ then
    add $(m, t-1, q)$ to $pending_p$
    if $t \geq ts_p[p]$ then
      $ts_p[p] \leftarrow t$
      send $(t, \perp)$ to all $q \neq p$
    fi
  fi
  $ts_p[q] \leftarrow t$
  delivery progress

delivery progress:
  repeat
    let $E = (m, t, r)$ be the minimum element in $pending_p$ by timestamp order
    if for some q, $ts_p[q] \leq E.t$ then exit fi
    deliver $E.m$
    remove E from $pending_p$
  end repeat

FIGURE 6. Algorithm Nonblocking-Broadcast

Observe that in algorithm Nonblocking-Broadcast-1, the communication of a broadcast message (x, m) is always immediately succeeded by the communication of timestamp message $(x+1)$ by the same process. We simplify Nonblocking-Broadcast-1 and simultaneously reduce the message traffic by replacing the two communications with only the one message $(x+1, m)$ which, upon receipt, is treated as both the broadcast message (x, m) and the timestamp message $(x+1)$ from the same process. This algorithm, called Nonblocking-Broadcast, is shown in figure 6. All messages sent are of the same type (x, m), where x is a timestamp, and m is a message which can be $\perp$. Message $(x, \perp)$ corresponds to a timestamp message in algorithm Nonblocking-Broadcast-1. The transformation is again straightforward and hence again the time complexity and the correctness of Nonblocking-Broadcast follows immediately from that of Algorithm Nonblocking-Broadcast-1.

Corollary 4.6. *Algorithm Nonblocking-Broadcast is a nonblocking atomic broadcast algorithm with time complexity* $2d$.

Since the time complexity of Nonblocking-Broadcast is the same as assumed by [1], the time complexities claimed therein for sequentially consistent implementations of data structures that use atomic broadcast follow.

5. Conclusions and future work

It is somewhat disturbing that nonblocking primitives may be needed for applications that are guaranteed to be blocking. In the instance studied in this paper, it appears that nonblocking atomic broadcast is required to efficiently implement blocking sequentially consistency algorithms for the specified model. On the other hand, a blocking primitive would have sufficed in a stronger model. It remains to analyze this phenomenon. Is there a general principle for translating from strong to weaker models at the cost of more powerful primitives? Is there a technique to limit the additional complications due to nonblocking?

References

1. H. Attiya and J. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994.
2. H. Attiya and J. Welch. Implementing *FIFO* queues and stacks. In *Proceedings of the 5th International Workshop, WDAG'91; LNCS No. 579; Springer-Verlag*, 1991.
3. K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

Appendix A. Decreasing timestamps can violate delivery order

Several simple modifications of Algorithm 1 to remove deadlock were considered. Under the assumption that atomic broadcast is blocking, the problem of deadlock can be removed by either (1) changing the test in line 4 to "if $t+1 \geq ts_p[p]$ then" or (2) swapping the order of the two instructions of *ABC-send*: a process first increases its timestamp, and then sends a message with the increased timestamp to all other processes. Unfortunately either of these ways to remove deadlock exposes other errors. In particular, if process p has received messages with higher timestamp between sending and receiving its own timestamp message, then this can cause the timestamp of process p to decrease. This, in turn, can jeopardize delivering the broadcasts in timestamp order, and as a consequence violates the condition that broadcasts are delivered to all processes in the same order.

A fragment of an execution of Algorithm 1 with lines 1 and 2 swapped that illustrates this phenomenon is given in figure 7. In this execution it is assumed that $u > (2d/3)$, and

$\epsilon < u - (2d/3)$. The order of process ids is $p < q < r$. Process p delivers broadcast message C with timestamp $(3, q)$ before broadcast message D with timestamp $(3, p)$ is sent. Process r delivers broadcast D before broadcast C, according to their timestamp order.

time	**process** p	**process** q	**process** r
0	$ABC\text{-}send(A)$: $ts_p[p] = 1$; send $(1, A)$	$ABC\text{-}send(B)$: $ts_q[q] = 1$; send $(1, B)$	
$d/3$		receive $(1, B)$ from q: $ts_q[q] = 2$; send ts(2)	receive $(1, B)$ from q: $ts_r[r] = 2$; send ts(2)
$2d/3$	receive $(1, A)$ from p: $ts_p[p] = 2$; send ts(2)	receive $(1, A)$ from p:	receive $(1, A)$ from p:
$5d/6$	receive $(1, B)$ from q: receive ts(2) from r: $ts_p[r] = 2$	receive ts(2) from q: $ts_q[q] = 2$ receive ts(2) from r: $ts_q[r] = 2$	receive ts(2) from q: $ts_r[q] = 2$ receive ts(2) from r: $ts_r[r] = 2$
d	receive ts(2) from q: $ts_p[q] = 2$ **deliver** A, B	receive ts(2) from p: $ts_q[p] = 2$ **deliver** A, B	receive ts(2) from p: $ts_q[p] = 2$ **deliver** A, B
$d + \epsilon$		$ABC\text{-}send(C)$: $ts_q[q] = 3$; send $(3, C)$	
$4d/3$	receive $(3, C)$ from q: $ts_p[p] = 4$; send ts(4)	receive $(3, C)$ from q: $ts_q[q] = 4$; send ts(4)	receive $(3, C)$ from q: $ts_r[r] = 4$; send ts(4)
$(5d/3) - \epsilon$	receive ts(4) from q: $ts_p[q] = 4$ receive ts(4) from r: $ts_p[r] = 4$ **deliver** C	receive ts(4) from q: $ts_q[q] = 4$	
$5d/3$	receive ts(2) from p: $ts_p[p] = 2$		
$(5d/3) + \epsilon$	$ABC\text{-}send(D)$: $ts_p[p] = 3$; send $(3, D)$		
$(5d/3) + 2\epsilon$	receive ts(4) from p: $ts_p[p] = 4$	receive ts(4) from p: $ts_q[p] = 4$	receive ts(4) from p: $ts_r[p] = 4$
$2d$	receive $(3, D)$ from p:	receive $(3, D)$ from p:	receive $(3, D)$ from p:
$7d/3$			receive ts(4) from q: $ts_r[q] = 4$ **deliver** D, C

FIGURE 7. Decreasing timestamp can violate the delivery order

If instead of swapping the two instructions in *ABC-send*, the test in line 4 is changed to "if $t + 1 \geq ts_p[p]$ then", an erroneous computation very similar to that in figure 7 can be constructed.

Another possible way to avoid deadlock is to always send a timestamp upon receipt of a broadcast message (that is, outdent line 6 in Algorithm 1). Figure 8 shows an erroneous execution of Algorithm 1 changed that timestamp message is sent unconditionally upon receiving a broadcast message (line 6 is outdented in Algorithm 1). As in the previous example it is assumed that $u > (2d/3)$, and $\epsilon < u - (2d/3)$. The order of process ids is $p < q < r$. Process p delivers message C with timestamp $(1, q)$ before message D with timestamp $(1, p)$ is ever sent. Process r delivers message D before message C, according to their timestamp order.

time	**process** p	**process** q	**process** r
0	$ABC\text{-}send(A)$: send $(0, A)$; $ts_p[p] = 1$	$ABC\text{-}send(B)$: send $(0, B)$; $ts_q[q] = 1$	
$d/3$		receive $(0, B)$ from q: send ts(1)	receive $(0, B)$ from q: $ts_r[r] = 1$; send ts(1)
$2d/3$	receive $(0, A)$ from p: send ts(1)	receive $(0, A)$ from p: send ts(1)	receive $(0, A)$ from p: send ts(1)
$5d/6$	receive $(0, B)$ from q: send ts(1) receive ts(1) from r:	receive ts(1) from q: receive ts(1) from r:	receive ts(1) from q: receive ts(1) from r:
d	receive ts(1) from q: receive ts(1) from q: receive ts(1) from r: **deliver** A, B	receive ts(1) from p: receive ts(1) from q: receive ts(1) from r: **deliver** A, B	receive ts(1) from p: receive ts(1) from q: receive ts(1) from r: **deliver** A, B
$d + \epsilon$		$ABC\text{-}send(C)$: send $(1, C)$; $ts_q[q] = 2$	
$4d/3$	 receive $(1, C)$ from q: $ts_p[p] = 2$; send ts(2)	receive ts(1) from p receive $(1, C)$ from q: send ts(2)	receive ts(1) from p receive $(1, C)$ from q: $ts_r[r] = 2$; send ts(2)
$d + \epsilon$		$ABC\text{-}send(C)$: send $(1, C)$; $ts_q[q] = 2$	
$4d/3$	 receive $(1, C)$ from q: $ts_p[p] = 2$; send ts(2)	receive ts(1) from p receive $(1, C)$ from q: send ts(2)	receive ts(1) from p receive $(1, C)$ from q: $ts_r[r] = 2$; send ts(2)
$(5d/3) - \epsilon$	receive ts(2) from q: receive ts(2) from r: **deliver** C	receive ts(2) from q:	
$5d/3$	receive ts(1) from p: $ts_p[p] = 1$ receive ts(1) from p:		
$(5d/3) + \epsilon$	$ABC\text{-}send(D)$: send $(1, D)$; $ts_p[p] = 2$		
$(5d/3) + 2\epsilon$	receive ts(2) from p:	receive ts(2) from p:	receive ts(2) from p:
$2d$	receive $(1, D)$ from p: send ts(2)	receive $(1, D)$ from p: send ts(2)	receive $(1, D)$ from p: send ts(2)
$7d/3$			receive ts(2) from q: **deliver** D, C

FIGURE 8. Delivery order can be violated even when timestamps are sent unconditionally