https://doi.org/10.1016/j.ipl.2019.105837

Crash-Tolerant Causal Broadcast in $O(n)$ Messages

Achour Mostéfaoui[°] Matthieu Perrin[°] Michel Raynal[◇,⋆], Jiannong Cao[⋆]

[°] LS2N, Université de Nantes, 44322 Nantes, France
[◇] Univ Rennes IRISA, Campus de Beaulieu, 35042 Rennes, France
[⋆] Department of Computing, Polytechnic University, Hong Kong

Abstract

Causal broadcast is a communication abstraction designed for asynchronous systems. It ensures that the messages broadcast by the processes are delivered in their broadcast causality order, namely, if the broadcast of a message m causally precedes the broadcast of a message m', no process delivers m' unless it has previously delivered m. Several algorithms implementing causal broadcast have been proposed for asynchronous systems prone to any number of process crashes. These algorithms rely on an underlying Reliable Broadcast abstraction, whose message cost is n^2. This paper presents a simple causal broadcast algorithm whose cost is n messages per causal broadcast. This is obtained at the cost of protocol messages whose size can be up to n application messages. Hence, the proposed algorithm is particularly interesting for applications whose messages are small.

Keywords: Asynchronous system, Backward recursion, Broadcast abstraction, Causal broadcast, FIFO message delivery, Message-passing, Process crash failure.

1 Introduction: Computing Model and Causal Broadcast

1.1 Computing Model

Process model The system is made up of a set of n processes denoted p_1, ..., p_n. Each process is sequential (which means it executes one step at a time), and asynchronous (which means it progresses at its own speed, which can vary with time and remains always unknown to the other processes).

Any number of processes can crash. A crash is a premature halt (after it crashed, if it ever does, a process executes no more steps). A process that does not crash is called *correct*. A process that crashes is called *faulty*.

Communication model Processes can send and receive messages through channels. It is assumed that any pair of processes is connected by a bidirectional channel. Channels are reliable and asynchronous. Reliable means that messages are neither corrupted, nor duplicated, nor lost. Asynchronous means that, albeit finite, the transit duration of a message is arbitrary.

The sending and the reception of a message are atomic steps. The processes can also use a broadcast macro-operation, denoted broadcast (msg) where msg is a message, which is a shortcut for "**for each** $j \in \{1, ...n\}$ **do** send (msg) to p_j **end for**". Let us observe that this macro-operation is not reliable, namely, if the sender crashes while executing it, an arbitrary subset of processes (possibly empty) receive the message.

A message created at the application level is called *application message*. A message generated by the algorithm implementing causal broadcast is called *protocol message*. When clear from the context, we use only the term *message*.

1.2 Causal Broadcast

Definition *Causal broadcast* was introduced by K. Birman and T. Joseph in a pioneering work on fault-tolerant distributed systems [3]. It is a communication abstraction that provides the processes with two operations denoted causal_broadcast() and causal_deliver(), that allow to broadcast and deliver messages. When a process invokes causal_broadcast(m), we say it c-broadcasts the message m; similarly when the invocation of causal_deliver() returns the message m, we say it c-delivers the message m (or m is c-delivered). Without loss of generality, it is implicitly assumed that all messages c-broadcast by the processes are different (this can be easily realized by associating a pair – local sequence number, sender identity – with each message, as done in this article). Formally, causal broadcast is defined by the following properties.

- Validity. If a process c-delivers a message m from a process p_i, p_i previously c-broadcast m.
- Integrity. A process c-delivers a message m at most once.
- Causal Delivery. If a process c-broadcasts a message m and later c-broadcasts a message m', or a process c-delivers a message m and later c-broadcasts a message m', no process c-delivers m' before c-delivering m.
- Termination. A message c-broadcast by a correct process is c-delivered by all correct processes.

Validity states that no spurious message can be c-delivered. Integrity states there is no message duplication. Causal Delivery states that the messages are c-delivered according to the causal links relating their c-broadcasts and c-deliveries [7]. Finally, Termination states that the correct processes c-deliver at least the messages they c-broadcast.

Example A simple example of a causal broadcast execution is presented at the left of Fig. 1. Because they are c-broadcast by the same process p_1, no process can c-deliver m_1 before m_0. The same holds for m_3 and m_4 which are c-broadcast by p_2: no process can c-deliver m_4 before m_3. As the c-broadcasts of m_2 and m_1 are not causally related, they can be c-delivered in a different order at distinct processes. The same holds for m_2 and m_3. Hence, the execution depicted in the figure satisfies the c-broadcast properties. Differently, if the message m_2 (i) had been delivered at p_2 before it issued the broadcast of m_3, and (ii) delivered before m_3 at process p_1, the execution would be incorrect. This is because m_3 would causally depend on m_2 and consequently no process would be allowed to deliver m_3 before m_2.

Figure 1: A causal broadcast execution and its message precedence graph

A short historical perspective As already said, causal broadcast was introduced by K. Birman and T. Joseph in [3]. Its first implementation was done in the ISIS system [2], where each message was required to carry its full causal past. A more efficient implementation was then introduced [4]. While it is now well accepted as a meaningful communication abstraction, causal broadcast initiated a controversy in the operating system community (see e.g., [1]). Several causal broadcast algorithms have been presented in the literature (e.g., see the textbooks [5, 11]), which use an underlying *reliable broadcast* abstraction.

Efficient implementations of causal broadcast for failure-free systems are presented in several articles and textbooks (e.g., [10, 12, 13, 14]). Formal characterizations of causal broadcast can be found in [6, 8, 10].

A more low-level approach was introduced in [9], where is defined the notion of a *conversation*, close to causal broadcast. The presentation of this paper is operating system-oriented and no precise definitions (i.e., properties-based) are given.

Motivation All the causal broadcast algorithms suited to crash-prone asynchronous message-passing systems we know are based on an underlying *reliable broadcast* abstraction. To ensure that each application message c-delivered by a correct process is c-delivered by all correct processes, all reliable broadcast algorithms use $O(n^2)$ protocol messages for each application message. However, while it is sufficient, it has never been proved that reliable broadcast was necessary to implement causal broadcast. Hence our investigation on the implementation of causal broadcast, which has given rise to the simple algorithm presented in this paper. This algorithm requires $O(n)$ protocol messages for the c-broadcast of an application message. This is obtained at the price of protocol messages sometimes carrying a few application messages (this depends on the application communication pattern). Hence, the proposed algorithm seems suited to the case where application messages are small. The principle that underlies the algorithm is the use of a precedence graph, which captures the causality relation on application messages. In some sense, the proposed algorithm can be seen as a variant of a reliable broadcast algorithm customized to causal broadcast.

Precedence graph Such a graph G is associated with each execution. Its vertices are the application messages. As far as its edges are concerned, there is a directed edge from a message m to a message m' if (i) the invocations of causal_broadcast(m) and causal_broadcast(m') are consecutive invocations by the same process p_i, or (ii) the c-delivery of m and the invocation of causal_broadcast(m') by a process p_j are consecutive in the sense there is no invocation of causal_broadcast$()$ by p_j between them. The figure depicts the precedence graph of the execution on its right. If the directed edge (m, m') belongs to G, we say that m' causally depends (directly) on m, or m is an immediate causal predecessor of m'. If there is a directed path from m to m', we say that m' depends (transitively) from m, or m is a causal predecessor of m. Two messages m and m' that are not causally related are said to be independent. The

causal past of a message m is the set of all the messages m' such that there is a directed path from m' to m.

2 An Efficient Causal Broadcast Algorithm

Algorithm 1, described below, implements causal broadcast. The messages that are c-broadcast are called *application* messages, while the messages generated by the algorithm are called *protocol* messages. As already indicated, this algorithm ensures that each invocation of causal_broadcast() generates n protocol messages.

Local variables at a process p_i Each process manages two local variables.

- sn_i is an integer variable, initialized to 0, used to associate a sequence number with each message c-broadcast by p_i.
- $copr_i$ (for *compressed predecessors* is a sequence of triplets $\langle m, j, sn\rangle$ such that the message m has been c-broadcast by process p_j with sequence number sn. A triplet $\langle m, j, sn\rangle$ is contained into $copr_i$ if p_i c-delivered m after its last c-broadcast and did not c-deliver any more recent message from p_j later (see Fig. 2).[1]

  The symbol ϵ denotes the empty sequence, and $\oplus$ denotes the concatenation of a new element at the end of the sequence.

Figure 2: Meaning of the sequence $copr_i$

Considering a process p_i, the meaning of $copr_i$ is illustrated on Fig. 2, in which p_i invoked first causal_broadcast(m_1) and later causal_broadcast(m_2). Moreover, between these two consecutive invocations, p_i c-delivered the messages m, m', and m'' (in this order). This means that, to ensure a correct message c-delivery order, (i) no process must c-deliver m_2 (i) before m_1 (because m_2 was c-broadcast later by the same process p_i), and (ii) before the messages m, m', and m'' (because their c-deliveries at p_i causally precede the c-broadcast of m_2). Hence, $copr_i$ is used to register all the messages c-delivered by p_i between any two consecutive invocations of causal_broadcast() it issues.

On the client side: the operation causal_broadcast (m) The code of this operation is pretty simple. Process p_i first increases sn_i, and then broadcasts the message $seqmsg$, which contains the sequence $copr_i \oplus \langle m, i, sn_i\rangle$ (lines 1-3). Let us notice that the triplet associated with the last message previously c-broadcast by p_i is suppressed from $copr_i$ before the broadcast of $seqmsg$. As p_i did not crash during the previous broadcast, this message is received by all processes, and the only ordering information needed is its sequence number, namely $(sn_i - 1)$, which is explicitly encoded in the triplet $\langle m, i, sn_i\rangle$.

Let us observe that any protocol message $seqmsg$ contains at least one triplet. The sequence of triplets $copr_i$ will be used as a causal barrier imposing to c-deliver all its messages before m. As just indicated, the sequence number sn_i will be used to ensure that the previous message c-broadcast by p_i is c-delivered before m.

[1] The timstamps associated with messages of $copr_i$ are the one that compose compressed vector clocks in [2]. Let us observe that the compressed predecessors of p_i include at most one messages from p_j if p_i, but can contain none of them. Said differently, $|copr_i| \leq n$.

```
operation causal_broadcast (m) is
(1)   sn_i ← sn_i + 1;
(2)   if ⟨−, i, −⟩ belongs to copr_i then suppress it from copr_i end if;
(3)   broadcast (seqmsg) where seqmsg = (copr_i ⊕ ⟨m, i, sn_i⟩);
(4)   copr_i ← ϵ. % empty sequence %

when seqmsg = ⟨m_1, i_1, sn_1⟩, · · · , ⟨m_ℓ, i_ℓ, sn_ℓ⟩ is received do
(5)   for x from 1 to ℓ do
(6)     if (m_x not yet c-delivered) then
(7)       wait(m, c-broadcast by p_{i_x} with seq. nb. (sn_x − 1), is c-delivered);
(8)       if ⟨m, i_x, sn_x − 1⟩ belongs to copr_i then suppress it from copr_i end if;
(9)       causal_deliver (m_x);
(10)      copr_i ← copr_i ⊕ ⟨m_x, i_x, sn_x⟩
(11)    end if
(12)  end for.
```

Algorithm 1: Efficient causal broadcast (code for p_i)

On the server side: Message reception When a process p_i receives a protocol message $seqmsg$ containing a sequence $\langle m_1, i_1, sn_1\rangle, \cdots, \langle m_\ell, i_\ell, sn_\ell\rangle$, it considers each of its triplet, one after the other ("for" loop, lines 5-12). Let $\langle m_x, i_x, sn_x\rangle$ the current triplet. It means that m_x was c-broadcast by p_{i_x}, and its sequence number is sn_x. There are two cases.

- If m_x has already been locally c-delivered, p_i proceeds to the next triplet, if any.
- If m_x has not yet been locally c-delivered, p_i first waits until the previous message (say m) c-broadcast by p_{i_x} is c-delivered (we can assume that each protocol message reception generates a new thread in charge of the triplets in $seqmsg$). This is made possible thanks to the sequence number sn_x (line 7). After this message m has been c-delivered, it is suppressed from $copr_i$ (line 8). This is due to the fact that the causal dependence between this message m and the next message c-broadcast by p_i must be updated, namely replaced by the new causal dependence between m_x and the next message c-broadcast by p_i (line 8). Finally, m_x can be c-delivered (line 10).

As we can see, each iteration step of the "for" loop (lines 5-12) has a backward recursive flavor, captured by the "wait" statement at line 7. This statement entails a recursive messages delivery, whose aim is to guarantee causal delivery.

3 Proof of the Algorithm

Lemma 1 *Each local variable $copr_i$ contains at most n triplets.*

Proof A local sequence $copr_i$ is initialized to the empty sequence ϵ, and reset to this value each time p_i invokes causal_broadcast(). Then, when p_i adds a new triplet $\langle m_x, i_x, sn_x\rangle$ to $copr_i$ (line 10), it previously suppresses the triplet $\langle -, i_x, sn_x - 1\rangle$ if it is in $copr_i$ (line 8). Hence, for any i_x, $copr_i$ contains at most one triplet $\langle -, i_x, -\rangle$, which proves the lemma. $\square_{Lemma\ 1}$

The next corollary is an immediate consequence of the previous lemma and lines 2-3.

Corollary 1 *A protocol message contains at most n triplets.*

Lemma 2 *Let us consider any two consecutive application messages m and m' c-broadcast by a process p_j. No process p_i c-delivers m' before m.*

Proof The proof is an immediate consequence of the "wait" statement of line 7, which imposes a FIFO delivery order between each pair of processes. $\square_{Lemma\ 2}$

Lemma 3 *Let us consider any two application messages m and m' such that* causal_broadcast(m) *causally precedes* causal_broadcast(m'). *No process c-delivers m' before m.*

Proof If m and m' are c-broadcast by the same process, the proof follows from Lemma 2. Let us now consider the case where m and m' are c-broadcast by different processes p_i and p_j. Let sn be the sequence number associated with m. As causal_broadcast(m) causally precedes causal_broadcast(m'), there is a directed path of messages in the precedence graph $m = m_1$, m_2, ..., $m_k = m'$, where $k \leq n-1$, and an associated sequence of distinct processes $p_{j(1)} = p_i, p_{j(2)}$, ..., $p_{j(k)} = p_j$ such that

- $m = m_1$ has been c-broadcast by $p_{j(1)} = p_i$ and c-delivered by $p_{j(2)}$ (in fifo order),
- after $p_{j(2)}$ c-delivered m_1, it c-broadcast m_2, which was c-delivered by $p_{j(3)}$ (in fifo order), etc.,
- after $p_{j(k)} = p_j$ c-delivered m_{k-1} (in fifo order), it c-broadcast $m' = m_k$.

Due to lines 7 and 9-10, and the definition of $seqmsg$ at line 3, it follows that, for any $x \in [1..k]$, $copr_{j(x)}$ includes, for each process p_y, the last causal predecessor of m_x c-broadcast by p_y (if any). Hence, when a process receives $seqmsg = \langle -,-,-\rangle, \cdots, \langle m', j, -\rangle$, the triplets in $seqmsg$ capture the messages that must be c-delivered by p_j before m', and one of these triplets is equal to $\langle -, i, sn''\rangle$ where $sn'' \geq sn$. It follows that p_j cannot c-deliver m' before m. $\square_{Lemma\ 3}$

Remark The previous lemma shows that, from a "basic principles" point of view, Algorithm 1 is a reduction of message causal delivery to (process-to-process) FIFO message deliveries.

Lemma 4 *If a correct process* (i) *c-broadcasts a message m, or* (ii) *c-delivers a message m and later c-broadcasts a message m', the message m is c-delivered by all correct processes.*

Proof Proof of (i). Let us assume by contradiction that there are application messages c-broadcast by correct processes that are never c-delivered by some correct process. Let m be a message and p_j its correct sender, such that m is the first message that is not c-delivered by a correct process p_i, while p_i c-delivers all its causal predecessor messages. This permanent blocking occurs at line 7. As p_j is correct, it broadcast a protocol message $seqmsg = (copr \oplus \langle m, j, sn\rangle)$. As m is the "first" (as defined above) message entailing the permanent blocking of p_i, all the messages in the prefix sequence $copr$, are c-delivered[2]. Consequently, the processing of all triplets in $copr$ terminates, and p_i processes then $\langle m, j, sn\rangle)$ (last iteration of the "for" loop). When this occurs, p_i remains blocked forever waiting for the previous message (say m') c-broadcast by p_j. But this permanent blocking means that m is not the first message for which p_i waits forever. This contradicts the assumption on m, and consequently, there is no "first" message from p_j never c-delivered by p_i.

Proof of (ii). If a process p_i c-delivers a message m from a (correct or crashed process) p_j, it adds the triplet $\langle m, k, sn\rangle$ (where p_k is the sender of m, and sn its sequence number) to $copr_i$ at line 10. Hence, this triplet will be broadcast when p_i will issue it next invocation of causal_broadcast(). $\square_{Lemma\ 4}$

Theorem 1 *Algorithm* 1 *implements the* Causal Broadcast *abstraction. Moreover, an invocation of* causal_broadcast() *costs n protocol messages, and each protocol message carries at most n application messages.*

[2]Let us observe that it is possible that the message $seqmsg$ broadcast by p_j carries a triplet $\langle m'', k, -\rangle$ such that p_k crashed while broadcasting the sequence of triplets $\langle -,-,-\rangle, \cdots, \langle m'', k, -\rangle$, and this sequence was never received by p_i. But in this case, if any, p_k did not crash during its previous invocation of broadcast() and consequently, p_i eventually receives the last message c-broadcast by p_k before m''.

Proof The Validity property follows directly from the text of the algorithm and the reliability of the network (none of them creates messages nor modifies their content). The Integrity property follows from line 6. The Causal Delivery property follows from Lemma 3 and the Termination property follows from part (i) of Lemma 4. The message cost properties follows from Lemma 1 and Corollary 1. $\square_{Theorem\ 1}$

4 Ensuring a Stronger Termination Property

A stronger termination for crashes during an invocation of broadcast() As already indicated, if a process crashes while executing broadcast($seqmsg$), some processes can receive $seqmsg$, while others do not. Let $P1$ be the former set of processes, and $P2$ the later. If the correct processes of $P1$ (if any) do not issue new invocations of causal_broadcast() (whose internal broadcast() will forward messages in $seqmsg$ to the processes of $P2$), the correct processes of $P2$ would never c-deliver messages in $seqmsg$ sent by processes that crashed during their last invocation of broadcast(). This raises the following question. In addition to the basic termination property, how to ensure the following stronger termination property:

- Strong Termination. If a correct process p_i c-delivers a message m, the message m is c-delivered by all correct processes.

This Strong Termination property ensures that all correct processes c-deliver the same set of messages. If the message m has been c-broadcast by a correct process, the basic termination property ensures strong termination. So, we have only to consider the case where the sender of m crashed while it was executing the unreliable macro-operation broadcast(). Hence, answering the question amounts to ensure that any message that (i) is broadcast by a process that crashes and (ii) is c-delivered by a correct process, is c-delivered by all correct processes.

A look at Lemma 4 To this end, we rely on part (ii) of Lemma 4. This lemma states more than the termination property, which demands only that, if a correct process c-broadcasts a message m, all correct processes c-deliver m. It additionally states that, if a correct process c-delivers a message m and later c-broadcasts a message m', the message m is c-delivered by all correct processes. In particular, if all correct processes repeatedly c-broadcast messages during an infinite execution, then the algorithm ensures strong termination.

From termination to strong termination It follows from the previous discussion that the strong termination property can be obtained as follows. Let p_i be a process that after some time does no longer invoke causal_broadcast(). This process is then required to c-broadcast a *control* application message (encoded in a default value, e.g., $\perp$), only if its variable $copr_i$ contains non-$\perp$ application messages. According to the algorithm, this invocation entails the required forwarding of the compressed predecessor application messages (which always succeeds if p_i is not a faulty process). [3]

A note on complexity For each message m c-broadcast by a process p_i, at most one message $\perp$ might be c-broadcast by any other process p_j. Therefore, in the worst case, this strategy might lead to the same number of messages as causal broadcast algorithms based on reliable broadcast. However, in practice, there are common situations in which the complexity would be much lower: (1) processes that

[3] A similar issue occurs when one has to solve early-deciding consensus in crash-prone synchronous message-passing systems. When a process attains a round r in which it knows the decision estimates of all the processes that were not crashed at the beginning of round r, it can compute the decision value and decide it. But, as it has no means to know if the other processes know the same set of estimate values, it must execute an additional round during which it propagates the decision value before terminating (see [11] for more details).

repeatedly c-broadcast messages never need to c-broadcast $\perp$ messages, and (2) one $\perp$ message can be used to forward up to $n-1$ messages. In particular, in an execution where processes first jointly c-broadcast and c-deliver $x \geq n$ messages, and only then c-broadcast one $\perp$ message each, $O(nx)$ protocol messages will be exchanged between processes, compared to $O(n^2x)$ messages for an algorithm based on reliable broadcast.

5 Conclusion

This article has presented a causal broadcast algorithm for asynchronous message-passing systems in which any number of processes can crash. Differently from causal broadcast algorithms based on reliable broadcast (which requires $O(n^2)$ protocol messages for each causal broadcast), the proposed algorithm generates only n protocol messages per causal broadcast. To this end, it replaces "early message forwarding" used in reliable broadcast by "late message forwarding", and reduces causal message delivery to fifo message delivery.

Acknowledgments

This work was partially supported by the French ANR project 16-CE40-0023-03 DESCARTES devoted to layered and modular structures in distributed computing. The authors want to acknowledge the referees for their constructive comments.

References

[1] Birman K.P., A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *Operating Systems Review*, 28(1):11-21 (1994)

[2] Birman K.P. and Cooper R., The ISIS project: real experience with a fault tolerant programming system. *Proc. 4th ACM SIGOPS European workshop, Operating Systems Review*, ACM Press, 25(2):103-107 (1991)

[3] Birman K.P. and Joseph T.A., Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47-76 (1987)

[4] Birman K.P., Schiper A., and Stephenson P., Lightweigt causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3): 272-314 (1991)

[5] Cachin Ch., Guerraoui R., and Rodrigues L., *Reliable and secure distributed programming*, Springer, 367 pages, ISBN 978-3-642-15259-7 (2011)

[6] Kshemkalyani A. and Singhal M., Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distributed Computing*, 11(2):91-111 (1998)

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

[8] Murty V.V. and Garg V.K., Characterization of message ordering specifications and protocols. *Proc. 7th Int'l Conference on Distributed Computer Systems (ICDCS'97)*, IEEE Press, pp. 492-499 (1997)

[9] Peterson L.L., Bucholz N.C., and Schilchting R.D., Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217-246 (1989)

[10] Raynal M., *Distributed algorithms for message-passing systems*. Springer, 510 pages, ISBN 978-3-642-38122-5 (2013)

[11] Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach. Springer*, 550 pages, ISBN: 978-3-319-94140-0 (2018)

[12] Raynal M., Schiper A., and Toueg S., The causal ordering abstraction and a simple way to simple to implement it. *Information Processing Letters*, 39(6):343-350 (1991)

[13] Schiper A., Eggli J., and Sandoz A., A new algorithm to implement causal ordering. *Proc. 3rd Int'l Workshop on Distributed Algorithms (WDAG'89)*, Springer LNCS 392, pp. 219-232 (1989)

[14] Schwarz R. and Mattern F., Detecting causal relationships in distributed computations: in search of the Holy Grail. *Distributed Computing*, 7:149-174 (1994)