Byzantine-Tolerant Causal Broadcast

Alex Auvolat[◇,†], Davide Frey[†], Michel Raynal[†,⋆], François Taïani[†]

[◇]École Normale Supérieure, Paris, France
[†]Univ Rennes, Inria, CNRS, IRISA, 35000 Rennes, France
[⋆]Department of Computing, Polytechnic University, Hong Kong

Abstract

Causal broadcast is a communication abstraction built on top of point-to-point send/receive networks, which ensures that any two messages whose broadcasts are causally related (as captured by Lamport's "happened before" relation) are delivered in their sending order. Several causal broadcast algorithms have been designed for failure-free and crash-prone asynchronous message-passing systems.

This article first gives a formal definition of a causal broadcast abstraction in the presence of Byzantine processes, in the form of two equivalent characterizations, and then presents a simple causal broadcast algorithm that implements it. The main difficulty in the design and the proof of this algorithm comes from the very nature of Byzantine faults: Byzantine processes may have arbitrary behavior, and the algorithm must ensure that correct processes (i) maintain a coherent view of causality and (ii) are never prevented from communicating between themselves. To this end, the algorithm is built modularly, namely it works on top of any Byzantine-tolerant reliable broadcast algorithm. Due to this modularity, the proposed algorithm is easy to understand and inherits the computability assumptions (notably the maximal number of processes that may be Byzantine) and the message/time complexities of the underlying reliable broadcast on top of which it is built.

Keywords: Algorithm, Asynchronous message-passing system, Byzantine process, Causal message delivery, Fault-tolerance, Modularity, Reliable broadcast, Simplicity.

1 Introduction

On causality Modern distributed systems have grown increasingly large, spanning millions of servers in data centers, and billions of devices scattered across the world. In this context, both theoreticians and practitioners have observed the impossibility of building highly available systems that offer strong consistency guarantees while tolerating network partitions [18, 20, 21]. As a result, a number of systems have chosen to rely on communication abstractions that offer causal-ordering guarantees (e.g. [30, 31, 44]). In the context of social networks, causal ordering ensures that users that see Bob's reply to Alice have also seen Alice's original message. In the context of collaborative editors, causal ordering enables large numbers of users to concurrently edit documents without conflicts [38]. In the domain of distributed data stores [15, 31], causal ordering makes automatic conflict resolution possible in the presence of concurrent writes [15].

In the last few years, multiple authors have observed that causal consistency and variants of it constitute the best achievable guarantee in large-scale fault-tolerant available systems [2, 31, 44]. It is therefore surprising that the majority of existing research work only considers and defines causal consistency in the context of crash failures.

Content of the paper This article first defines a Byzantine-tolerant causal ordering abstraction (which we call BCO-broadcast). Then it presents (and proves correct) an algorithm implementing it in a system where communication is by message-passing. This algorithm is built on top of a Byzantine reliable broadcast abstraction. Consequently it inherits the t-resilience and the message/time complexities of the specific algorithm chosen to implement the underlying reliable broadcast. While the algorithm is simple, its proof is not, due the very nature of Byzantine faults. Namely, the main difficulty consists in proving that Byzantine processes can neither destroy the causality relation on the messages broadcast by the correct processes nor prevent the correct processes from delivering (according to causal order) the messages they broadcast.

Roadmap The article is composed of 8 sections. Section 2 presents some background and related work. Section 3 presents the computation model and Byzantine reliable broadcast. Section 4 defines Byzantine causal broadcast and presents a property-based characterization of it. Section 5 presents the Byzantine causal broadcast algorithm. Section 6 presents its proof (which is based on the previous characterization theorem). Section 7 presents an example of use of Byzantine causal broadcast. Finally, Section 8 concludes the paper.

2 Background and Related Work

On reliable broadcast *Reliable broadcast* [23] is a communication abstraction central to fault-tolerant asynchronous distributed systems. It allows each process to broadcast messages in the presence of process failures, with well-defined delivery properties. More precisely, it guarantees that non-faulty processes deliver the same set of messages, which includes at least all the messages they broadcast. This set may also contain messages broadcast by faulty processes. The fundamental property of reliable broadcast lies in the fact that no two non-faulty processes deliver different sets of messages.

Reliable broadcast in the presence of Byzantine processes (BR-broadcast) Reliable broadcast has been studied in the context of Byzantine failures since the eighties [7]. A process commits a Byzantine failure if it behaves arbitrarily (i.e., its behavior is not the one described by the algorithm it is assumed to execute) [29]. Such a failure can be intentional (also called malicious) or the result of transient faults which altered the content of some local variables at some processes, thereby modifying their intended behavior in unpredictable ways.

An elegant signature-free algorithm, which implements the reliable broadcast abstraction in n-process asynchronous systems where the processes communicate by message-passing and up to $t < n/3$ of them may be Byzantine was proposed by Bracha [7]. Bracha's algorithm is optimal with respect to t-resilience, since $t < n/3$ is an upper bound on the number of Byzantine processes that can be tolerated [8, 39]. From an operational point of view, this algorithm is based on a "double echo" mechanism of the value broadcast by the sender process. For each application message[1] it uses three types of protocol messages, requires three consecutive communication steps (one for each message type), and generates $(n-1)(2n+1)$ protocol messages. Another Byzantine reliable broadcast algorithm has recently been introduced in [26]. This algorithm implements the reliable broadcast of an application message with only two consecutive communication steps, two message types, and $n^2 - 1$ protocol messages. The price to pay for this gain in efficiency is a weaker t-resilience, namely $t < n/5$. Hence, these two algorithms differ in their trade-offs between t-resilience and message/time efficiency.

[1]An *application message* is a message sent by the reliable broadcast abstraction, while a *protocol message* is a message used to implement reliable broadcast.

Causal order broadcast Causal order broadcast is a communication abstraction which adds "quality of service" on top of reliable broadcast. Denoted CO-broadcast (where CO stands for Causal Order), it was introduced by K. Birman and T. Joseph in a pioneering work on fault-tolerant distributed systems [5]. This seminal work, further discussed in [3, 4], considers process crash failures.

CO-broadcast states that any two messages whose broadcasts are causally related (according to Lamport's *happened before* relation [28]), are delivered in their causal sending order. Messages whose broadcast are not causally related can be delivered in different orders at different processes. From a causality point of view, CO-broadcast extends the FIFO property–which considers each channel separately–to system-wide causality. Like FIFO broadcast, CO-broadcast constitutes a multi-shot communication abstraction (namely, all the invocations of CO-broadcast are related by the same causality relation).

Theoretical characterizations of CO-broadcast can be found in [27, 36, 39]. Algorithms implementing this communication abstraction can be found in [41, 43] for failure-free asynchronous message-passing systems, and in [6, 9, 35, 39] for asynchronous message-passing systems where any number of processes may crash.

Although, to the best of our knowledge, this paper is the first to propose a Byzantine Causal Broadcast primitive, we believe this contribution chimes in with the recent renewed interest in consensus-free Byzantine objects, which have recently been shown to be particularly interesting in the context of cryptocurrencies [22], trustless eventually consistent distributed objects, and commutative replicated data types [44].

Limit of causal broadcast In the context of Byzantine failures, it is possible for a Byzantine process that receives a protocol message carrying an application message to read the content of this application message, despite the fact that it might not yet have been bco-delivered. This creates a kind of "insider trading" behavior, which cannot be solved by BCO-broadcast alone. This problem was first addressed in [42]. Up to now, this notion of secure causality has been introduced only for total order broadcast ([42] with further developments in [10, 16]). More generally, we observe that, like for consensus, the definition of BCO-broadcast may need to be customized for the specific problem being solved.[2]

3 Computation Model

3.1 On the process side

Asynchronous processes The system is made up of a finite set Π of $n > 1$ asynchronous sequential processes, namely $\Pi = \{p_1, \ldots, p_n\}$. *Asynchronous* means that each process proceeds at its own speed, which can vary arbitrarily with time, and always remains unknown to the other processes.

Process failure Up to t processes can exhibit a *Byzantine* behavior [29]. A Byzantine process is a process that behaves arbitrarily: it can crash, fail to send or receive messages, send arbitrary messages, start in an arbitrary state, perform arbitrary state transitions, etc. As a simple example, a Byzantine process, which is assumed to broadcast a message m to all the processes, can send a message m_1 to some processes, a different message m_2 to another subset of processes, and no message at all to the remaining processes. Moreover, Byzantine processes can collude to foil non-Byzantine processes. As is

[2]The validity property of consensus states which value can be decided. Its definition depends on the failure model. In asynchronous crash-prone systems, validity requires the decided value to be a proposed value (which consequently can be the value proposed by a faulty process). In the Byzantine context, several validity properties have been proposed. As an example one states that if all the correct processes propose the same value, then this value must be decided (in this case, if not all the correct processes propose the same value, any value can be decided –including a value that has not been proposed– [32]). A different validity property states that the decided value must not be a value proposed by Byzantine processes only, namely the decided value must be either a value proposed by a correct process (and this must always occur when at least $t + 1$ correct processes propose the same value) or a predefined default value [13, 25, 39]).

common in Byzantine-Fault-Tolerant algorithms, we assume that Byzantine processes cannot spawn or impersonate other processes, i.e., we exclude Sybil attacks. A process that exhibits a Byzantine behavior is also called *faulty*. Otherwise, it is *correct* or *non-faulty*.

Let us notice that, as each pair of processes is connected by a channel, a process can identify the sender of each message it receives. Hence, no Byzantine process can impersonate another process.

3.2 On the communication side

The Byzantine reliable broadcast communication abstraction The BR-broadcast abstraction is a one-shot communication abstraction that provides each process with two operations, br_broadcast() and br_deliver(). *One-shot* means that a process invokes br_broadcast() at most once, and BR-broadcast instances issued by processes are independent. As in [9, 23, 39], we use the following terminology: when a process invokes br_broadcast(), we say that it "br-broadcasts a message", and when it executes br_deliver(), we say that it "br-delivers a message". BR-broadcast is defined by the following properties.

- BR-Validity. If a correct process br-delivers a message m from a correct process p_i, then p_i br-broadcast m.
- BR-Integrity. A correct process br-delivers at most one message m from a process p_i.
- BR-Termination-1. If a correct process br-broadcasts a message, it br-delivers it.
- BR-Termination-2. If a correct process br-delivers a message m from p_i (possibly faulty) then all correct processes eventually br-deliver m from p_i.

On the safety side, BR-validity relates the outputs (messages br-delivered) to the inputs (messages br-broadcast), while BR-integrity states that there is no duplication.

On the liveness side, BR-Termination-1 states that a correct process br-delivers the messages it has br-broadcast, while BR-Termination-2 gives its name to reliable broadcast. Be the sender correct or not, every message br-delivered by a correct process is br-delivered by all correct processes. It follows from these properties that all correct processes br-deliver the same set of messages, and this set contains at least all the messages br-broadcast by the correct processes.

As indicated in Section 2, there are signature-free distributed algorithms, which build the BR-broadcast communication abstraction on top of asynchronous message-passing systems in which processes may be Byzantine [7, 26, 39].

From one-shot to multi-shot BR-broadcast BR-broadcast is a one-shot communication abstraction: it allows a given process to invoke the operation br_broadcast() only once. Hence, the BR-broadcast instance invoked by p_i can be identified by the process index i.

A simple way to obtain a more general multi-shot BR-broadcast abstraction (MBR-broadcast) consists in associating a specific tag with each invocation of br_broadcast() by a process [9, 39]. Implementing the tags with sequence numbers, a process p_i now invokes br_broadcast(sn_i, m), where sn_i is the current value of the local integer variable used by p_i to generate its sequence numbers. This instance is then identified by the pair $\langle i, sn_i \rangle$.

For the MBR-broadcast abstraction the BR-Validity and BR-Integrity property become:

- MBR-Validity. If a correct process br-delivers a message m from a correct process p_i with sequence number sn, then p_i br-broadcast m with sequence number sn.
- MBR-Integrity. Given a sequence number sn, a correct process br-delivers at most one message m associated with sn from a process p_i.

The properties MBR-Termination-1 and MBR-Termination-2 are the same as their BR-broadcast counterparts, with the addition that sequence numbers are also preserved. Note that at this stage we are not assuming a relationship between sequence numbers and order of emission or of reception of messages. Multi-shot extensions of the one-shot signature-free BR-broadcast algorithms introduced in [7, 26] are presented in Appendix A.

Invocation pattern To simplify both the understanding and the presentation of the algorithm implementing the BCO-broadcast abstraction, the invocation previously written $\mathsf{br\_broadcast}(sn_i, m)$ by process p_i is replaced by $\mathsf{br\_broadcast}(\langle i, sn_i\rangle, m)$ (where i must be considered as a comment). This makes explicit for the reader the identity of the corresponding MBR-broadcast instance.

4 Byzantine Causal Order Broadcast: Definition and a Characterization

This section first proposes a definition of Byzantine causal order broadcast (BCO-broadcast). This definition is actually an extension of the MBR-broadcast abstraction with a new property called Byzantine Causal Order, that defines a partial order on the set of application messages. Then, this section presents a theorem which characterizes BCO-broadcast with a different set of properties which correspond more closely to the algorithm. This theorem will be used in Section 6 to prove the algorithm described in Section 5.

4.1 Definition

BCO-broadcast is a multi-shot communication abstraction, which provides processes with the operations denoted $\mathsf{bco\_broadcast}()$ and $\mathsf{bco\_deliver}()$ (hence a message is “bco-broadcast” and “bco-delivered”). Similarly to the case of crash-failures, BCO-Broadcast must capture causality in the entire system. This distinguishes it from FIFO-Broadcast which only requires that, for each sender process, messages from this process are delivered in their sending order. We will now give a formal characterization of this requirement.

Let BCO-Validity, BCO-Integrity, BCO-Termination-1, and BCO-Termination-2 denote the same properties as their MBR-broadcast counterparts. The definition of BCO-broadcast is based on the following partial order defined on the (application) messages.

Byzantine causal order Let M be the set of (application) messages bco-delivered by correct processes in an execution E of an algorithm $\mathcal{A}$, which respects the BCO-Validity, BCO-Integrity, BCO-Termination-1, and BCO-Termination-2 properties. A *Byzantine causal order* on M is a partial order $\rightarrow_M$ on M such that:

- BCO-1. For any (correct or faulty[3]) process p_i, the set of messages from p_i bco-delivered by correct processes is totally ordered by $\rightarrow_M$.
- BCO-2. If p_i is correct and bco-delivered or bco-broadcast a message m before bco-broadcasting a message m', then $m \rightarrow_M m'$.

A Byzantine causal order $\rightarrow_M$ captures the causality relation on the set of messages bco-delivered by the correct processes. This set of messages, which includes all the messages bco-broadcast by the correct

[3]Faulty nodes are included in BCO-1 for the same reason they are in property BR-Termination-2 (Section 3.2). Correct processes do not know beforehand who is Byzantine. Including Byzantine processes in BCO-1 (and in BR-Termination-2) ensures all correct nodes interpret a Byzantine behavior in the same way, while circumventing the thorny problem of detecting and agreeing on who is Byzantine.

processes, may also include messages from Byzantine processes[4]. BCO-1 ensures the minimal property that all correct processes agree on a same delivery order for the messages they bco-deliver from a given (correct or Byzantine) process. BCO-2 establishes the ordering relationship between the messages bco-broadcast by a correct process and the messages it previously bco-delivered or bco-broadcast.

Byzantine causal order reliable broadcast (BCO-broadcast) The BCO-broadcast abstraction is defined by the BCO-Validity, BCO-Integrity, BCO-Termination-1, and BCO-Termination-2 properties, plus the following causality-related property.

- BCO-Causality. There is a Byzantine causal order $\rightarrow_M$ on the set M, such that, if for any pair of messages $m, m' \in M$, $m \rightarrow_M m'$, then no correct process bco-delivers m' before m.

Remark that, by the combination of BCO-1 and BCO-2, BCO-Causality states that a correct process must bco-deliver the messages from any other correct process in their sending order.

4.2 A local order property

The aim of this section is to state a theorem giving a characterization of BCO-broadcast. Following the approach introduced in [23] for crash failures, this equivalence will be instrumental in proving the algorithm implementing the BCO-broadcast abstraction. To this end, let us consider the two following properties on message delivery.

- BCO-Fifo-order. If a correct process p_i bco-delivers two messages m and m' from the same process p_k in the order first m and then m', no correct process bco-delivers m' before m (BCO-Fifo-1). Moreover, if p_k is correct, it bco-broadcast m before m' (BCO-Fifo-2).
- BCO-Local-order. If a correct process bco-delivers first a message m and later bco-broadcasts a message m', no correct process delivers m' before m.[5]

The two components of the BCO-Fifo-order property can be understood as follows: BCO-Fifo-1 states that no two correct processes disagree on the delivery order of the messages from the same process p_k (whether p_k is correct or Byzantine). BCO-Fifo-2 states that, if the sender is correct, the delivery order is the same as the sender's broadcast order.

The theorem that follows states a strong equivalence relating BCO-Causality on one side and BCO-Fifo-order plus BCO-Local-order on the other side:

Theorem 1. *Under the assumptions of the properties* BCO-Validity, BCO-Integrity, BCO-Termination-1*, and* BCO-Termination-2*, the property* BCO-Causality *is equivalent to* BCO-Local-order *and* BCO-Fifo-order.

Proof It can be easily seen that BCO-Causality (through BCO-1 and BCO-2) implies the BCO-Fifo-order property as well as the BCO-Local-order property.

In the other direction, let $\mathcal{A}$ be an algorithm that satisfies the BCO-Local-order and BCO-Fifo-order properties. Let M be the set of messages bco-delivered by correct processes during an execution of $\mathcal{A}$. We have to show that M satisfies BCO-Causality, namely, there is a partial order $\rightarrow_M$ that is is included in the (total) bco-delivery order of any correct process, and that satisfies BCO-1 and BCO-2. Let us define the relation $\rightarrow_M$ as follows:

A If any correct process bco-delivered a message m before a message m', both from the same sender process, then $m \rightarrow_M m'$.

[4]Let us remind that, while a Byzantine process can invoke bco_broadcast() without executing its code, it can also send (correct of fake) messages (which appear in the $\rightarrow_M$ relation and are consequently bco-delivered by the correct processes) without having invoked the bco_broadcast() operation.

[5]This property was introduced in [23] in the context of crash-prone asynchronous systems.

B If any correct process bco-delivered a message m before bco-broadcasting m', then $m \rightarrow_M m'$.

C If $m \rightarrow_M m'$ and $m' \rightarrow_M m''$, then $m \rightarrow_M m''$.

Let us now show that, for any correct process p_i, the relation $\rightarrow_M$ is included in the (total) bco-delivery order at p_i.

- Case $m \rightarrow_M m'$ because of (A). In this case, m and m' are from the same sender and there exists a correct process that bco-delivered them in the order first m, then m'. Therefore, due to the BCO-Termination-2 and the BCO-Fifo-order properties, p_i bco-delivers them in the same order.
- Case $m \rightarrow_M m'$ because of (B). In this case, due to the BCO-Termination-1, BCO-Termination-2, and BCO-Local-order properties, p_i bco-delivered m before m'.
- Case $m \rightarrow_M m'$ because of (C). The local bco-delivery order at p_i is a total order, therefore transitively closed. Hence, if $m \rightarrow_M m'$ due to (C), by induction, p_i bco-delivered m before m'.

It follows from the previous items that $\rightarrow_M$ does not contain cycles, and is consequently a partial order. Let us now show that the partial order $\rightarrow_M$ satisfies the properties BCO-1 and BCO-2.

- BCO-1. Let M_i be the set of messages bco-delivered from a process p_i at a correct process. It follows from BCO-Termination-2 that any correct process bco-delivered them. Moreover, due to (A), any pair of messages m and m' of M_i is ordered by $\rightarrow_M$. It follows that, for any (correct or faulty) process p_i, the set of messages bco-delivered from p_i by correct processes is totally ordered by $\rightarrow_M$, which is the property BCO-1.
- BCO-2. First, if a correct process bco-delivers m before bco-broadcasting m', due to (B) we have $m \rightarrow_M m'$. Second, if a correct process bco-broadcast m before bco-broadcasting m', due to the BCO-Termination-1, BCO-Termination-2, and BCO-Fifo properties, all correct processes bco-deliver m before m', hence, due to (A) we have $m \rightarrow_M m'$. Combining the two cases, if a correct process p_i bco-delivered or bco-broadcast a message m before bco-broadcasting a message m', we have $m \rightarrow_M m'$, which is the statement BCO-2.

Thus $\rightarrow_M$ is a Byzantine causal order on M and all correct processes bco-deliver the messages of M in an order that contains $\rightarrow_M$, which concludes the proof of the theorem. $\Box_{Theorem\ 1}$

5 BCO-broadcast on Top of MBR-broadcast: an Algorithm

While at this point the characterization of a causal order in Byzantine systems is well understood, what remains to be shown is that an algorithm that implements such an ordering and that terminates is possible. This is done in this section and the following, in which we respectively introduce Algorithm 1 that implements the BCO-broadcast abstraction and prove its correctness. This algorithm is based on the simple notion of a *causal barrier* [23], which we track internally using a vector clock [17, 34, 45]. While the statement of this algorithm is close to the one of existing failure-free or crash-tolerant causal broadcast algorithms, its proof (given in Section 6) is far from being a simple extension of the corresponding proofs. This heightened difficulty comes from

- the fact that, while in the crash-failure model, a process behaves correctly until it possibly crashes, a Byzantine process can exhibit an arbitrary behavior at any time, and
- the algorithm must ensure that correct processes (i) are never prevented from communicating between themselves, and (ii) always maintain a coherent view of message causality.

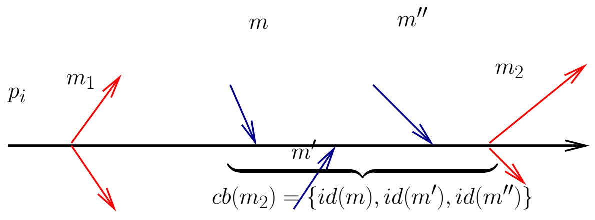

Figure 1: Meaning of the set cb_i

Local data structures Each process manages the following local variables.

- sn_i is an integer (initialized to 0) used by p_i to associate a sequence number with the messages it co-broadcasts.
- $co\_del_i[1..n]$ is an array of integers such that $co\_del_i[j]$ counts the number of messages that p_i has bco-delivered from p_j.
- cb_i (where cb stands for "causal barrier") is a set initialized to $\emptyset$, whose meaning is explained below.

Basic principle underlying the algorithm: capturing causal barriers Let us consider the example given in Fig. 1, where p_i first invoked bco_broadcast(m_1) and later invoked bco_broadcast(m_2). Moreover, between these two consecutive invocations, m, m' and m'' are the messages bco-delivered by p_i, and these messages are not causally related.

This means that, to ensure a correct bco-delivery of m_2, a correct process p_k is (i) neither allowed to bco-deliver m_2 before m_1 (because m_2 was bco-broadcast after m_1 by the same process p_i), (ii) nor allowed to bco-deliver m_2 before the messages m, m' and m'' (because their bco-deliveries at p_i causally precede the bco-broadcast of m_2). Hence, p_i stores in cb_i the identity of the messages it bco-delivers (after the bco-broadcast of m_1) that are immediate causal predecessors of the next message it will bco-broadcast in the future (here m_2).

Let us remark that, if after the bco-broadcast of m_1 and before the bco-broadcast of m_2, p_i bco-delivers a message m^* that causally precedes m^+ (where m^+ is m or m' or m''), the identity of m^* is first saved in cb_i and later suppressed from it when m^+ is bco-delivered. As already said, cb_i contains the identities of the messages that are *immediate* predecessors (from a message causality point of view) of the next message that p_i will bco-broadcast.

Notations We use the following notations. Let m be an application message bco-delivered by a correct process, and $\langle i, sn\rangle$ its identity (i.e., the identity of its BCO-broadcast instance).

- $id(m)$ returns the identity of m, namely the pair $\langle i, sn\rangle$.
- $msg(i, sn)$ returns the message m identified by $\langle i, sn\rangle$.
- $cb(m)$ returns the identities of the messages that define the causal barrier of m. In Fig. 1, we have $cb(m_2) = \{id(m), id(m'), id(m'')\}$.

Crash failures vs Byzantine Failures: validity predicate In the crash failure model, a process always executes correctly its algorithm until it terminates or crashes. Crashes are "syntactic" in the sense that they are only due to the environment in which the application executes. The situation is different in the case of Byzantine faults which can be due not only to the environment but also to the fake data communicated by Byzantine processes. As a result, in a Byzantine context, correct processes might need to check the validity of an incoming message in terms of application logic before delivering this message. Although such validity checks are application specific, they need to be injected within the causal broadcast algorithm. This *dependency injection* [19] allows correct processes to ensure that the causal past of bco-delivered messages only includes messages that are valid in terms of application semantics. This ability to ignore invalid messages while tracking causality is a natural requirement in many applications in order to curtail the disruptive power of Byzantine processes. Hence, according to the application and if needed, an appropriate predicate must allow processes to check the validity of the data contained in the messages they receive.

Following an idea introduced in [10] and used in [14], we assume an application-dependent predicate to address this issue. This predicate must be provided by application developers and invoked from within the implementation of the BCO-broadcast abstraction to control or even prevent the bco-delivery of invalid messages[6]. This predicate is named $\mathsf{validity\_pred}(j, m)$ in the BCO-broadcast algorithm that follows. Its parameters j and m are the identity of the sender process and the application message to be bco-delivered, respectively. For the applications based on BCO-broadcast that do not need such a predicate, it is assumed that it always returns true. An example of the use of this predicate is given in Section 7.2.

```
init $cb_i \leftarrow \emptyset$; $sn_i \leftarrow 0$; $co\_del_i \leftarrow [0, \ldots, 0]$.

operation bco_broadcast$(m)$ at $p_i$ is
(1)  $sn_i \leftarrow sn_i + 1$;
(2)  br_broadcast$(\langle i, sn_i\rangle, \langle cb(m), m\rangle)$ where $cb(m) = cb_i$;
(3)  $cb_i \leftarrow \emptyset$;
(4)  return().

when $(\langle j, sn\rangle, \langle cb(m), m\rangle)$ is br_delivered from $p_j$
(5)  wait$\big((sn = co\_del_i[j] + 1) \wedge (\forall\, \langle k, sn'\rangle \in cb(m) :\ co\_del_i[k] \geq sn')$
              $\wedge(\mathsf{validity\_pred}(j, m))\big)$;   % application-defined predicate
(6)  $cb_i \leftarrow (cb_i \setminus cb(m)) \cup \{\langle j, sn\rangle\}$;
(7)  local bco_delivery of $m$ from $p_j$;
(8)  $co\_del_i[j] \leftarrow co\_del_i[j] + 1$.
```

Algorithm 1: BCO-broadcast on top of MBR-broadcast (code for p_i)

Algorithm The algorithm is quite simple. When a process p_i invokes $\mathsf{bco\_broadcast}(m)$, it calls the underlying $\mathsf{br\_broadcast}()$ operation with the protocol message $\langle cb(m), m\rangle$ and sequence number sn_i, where $cb(m)$ is the current value of cb_i. Then, it resets cb_i to the empty set.

When it br-delivers the message $\langle cb(m), m\rangle$ identified $\langle j, sn\rangle$ from p_j, p_i waits until the associated delivery condition becomes satisfied. This condition states that the previous message bco-broadcast from p_j and all the immediate causal predecessors of m have been locally bco-delivered (which is operationally captured by the predicate $\forall\, \langle k, sn'\rangle \in cb(m) :\ co\_del_i[k] \geq sn'$ (line 5)).

When this occurs, p_i updates cb_i to reflect the causality links created by the bco-delivery of m, namely, it suppresses from cb_i the message identities that are in $cb(m)$ (as these messages are no longer immediate predecessors of the next message m' that p_i will bco-broadcast), and replaces them by the identity of m (line 6). Process p_i then bco-delivers m from p_j and accordingly increases $co\_del_i[j]$.

Trivially, the computability assumption and the messages/time complexities of Algorithm 1 are the ones of the underlying Byzantine reliable broadcast algorithm it uses.

[6]From an operational point of view, this is similar to the *upcall* mechanism found in operating systems, be them centralized [11] or distributed [12].

6 Proof of the Algorithm

This section first shows that Algorithm 1 satisfies the characterization properties introduced in Section 4.2. The proof that the algorithm implements BCO-broadcast then follows from Theorem 1.

The proof consists in showing that, while correct processes may bco-deliver messages from Byzantine processes (and consequently these messages participate in the causality relation), they cannot prevent the correct processes from bco-delivering all the messages they bco-broadcast. Hence the central parts are the proof of the Termination properties (see Theorem 2), and the proof of BCO-Local-order property (see Theorem 3).

As the predicate $\mathsf{validity\_pred}()$ is application-dependent, the proofs of the BCO-Termination-1, BCO-Termination-2 and BCO-Causality properties assume that the predicate $\mathsf{validity\_pred}()$ always returns true. In the general case, the validity predicate $\mathsf{validity\_pred}()$ offers application developers a broad control of the termination properties of the BCO-broadcast algorithm. More precisely, a validity predicate conserves the termination properties of BCO-broadcast if (i) correct processes ensure that the predicate is true for the messages they broadcast, and (ii) a predicate that is true for a message m can only be invalidated by a message from the same sender as m. In this case, the FIFO-delivery of messages and the reliability of the underlying MBR-broadcast used by BCO-broadcast continue to guarantee the termination of BCO-broadcast. As to the causal delivery order of messages, the validity predicate has no influence on the BCO-Causality property of the algorithm, as it can only be used to introduce new delivery constraints and not to cancel the causal order delivery constraints coded by the wait statement of line 5.

Theorem 2. *Algorithm 1 satisfies the* BCO-Validity, BCO-Integrity, BCO-Termination-1, *and* BCO-Termination-2 *properties.*

Proof Let us remember that the underlying MBR-broadcast abstraction ensures that all correct processes br-deliver the same set of messages and this set includes at least the messages they br-broadcast.

- Proof of the BCO-Validity property. This property directly follows from MBR-Validity. This is because for a message m to be bco-delivered (by a correct process) from a correct process p_i (line 7), it has to be br-delivered from that same process, meaning it was br-broadcast by p_i at line 2) and consequently p_i bco-broadcast m.
- Proof of the BCO-Integrity property. The proof follows the fact that each message m that is bco-delivered by a correct process p_i has an identity $\langle j, sn\rangle$, and MBR-Integrity guarantees that no two messages are br-delivered with the same identity. Consequently, p_i cannot bco-deliver another message m' from p_j with the same identity.
- Proof of the BCO-Termination-1. It follows from the bco-delivery procedure (lines 5-8) that, for any correct process p_i and at any time, we have $\forall\ \langle j, sn\rangle \in cb_i, co\_del_i[j] \geq sn$. Therefore, when a correct process p_i bco-broadcasts a message m with sequence number sn, it br-broadcast $(\langle i, sn\rangle, \langle cb(m), m\rangle)$ (line 2), and will br-deliver it (by MBR-termination-2), and the middle term of the wait condition (line 5) is already verified. The first term of the wait condition imposes that the process's own messages are bco-delivered in FIFO order: it becomes true for the first sequence number not yet bco-delivered. Since all sequence numbers are br-delivered, all messages from p_i preceding m (if any) and m itself are eventually bco-delivered by p_i.
- Proof of the BCO-Termination-2. Let p_i be a correct process, and $m_1, m_2, \ldots$ the sequence of messages bco-delivered by p_i. We show by induction the following property: if p_i bco-delivers all the messages $m_1, \ldots, m_k$, then all correct processes bco-deliver all the messages $m_1, \ldots, m_k$.
  - Base case: $k = 0$. The property is true because the set of messages bco-delivered by p_i is then empty.

- General case. Suppose that, for a given $k > 0$, all correct processes bco-delivered $m_1, \dots, m_k$. We need to show that all correct processes bco-deliver m_{k+1}. When p_i bco-delivers m_{k+1} the wait condition at line 5 is verified at p_i with the following values (where $sn(m)$ and $proc(m)$ are the sequence number and the sender process of m, respectively, and $\mathsf{max}\{\} = 0$):

$$co\_del_i[\ell] = \mathsf{max}\big\{sn(m) \text{ such that } m \in \{m_1, \dots, m_k\} \wedge proc(m) = \ell\big\}. \quad (1)$$

Let p_j be another correct process. After it bco-delivered $m_1, \dots, m_k$, a similar property is verified for p_j:

$$co\_del_j[\ell] \geq \mathsf{max}\big\{sn(m) \text{ such that } m \in \{m_1, \dots, m_k\} \wedge proc(m) = \ell\big\}. \quad (2)$$

In equation (2), "$=$" is replaced by "$\geq$" since p_j may have bco-delivered messages other than $m_1, \dots, m_k$. This equation remains true in subsequent steps, as $co\_del_j[\ell]$ can only increase. Thus after p_j bco-delivered $m_1, \dots, m_k$ and br-delivered m_{k+1}, the second term of the wait condition of line 5 is true.

Let us note $p_s = proc(m_{k+1})$ the sender of m_{k+1}. When p_i bco-delivers m_{k+1}, we also have $sn(m_{k+1}) = co\_del_i[s] + 1$, which combined with (1) yields

$$sn(m_{k+1}) = \mathsf{max}\big\{sn(m) \text{ such that } m \in \{m_1, \dots, m_k\} \wedge proc(m) = s\big\} + 1. \quad (3)$$

There is a moment when p_j has bco-delivered $m_1, \dots, m_k$ and has br-delivered m_{k+1} but not yet bco-delivered it. At this moment, since by MBR-Integrity m_{k+1} is the only message with sequence number $sn(m_{k+1})$ from p_s that p_j ever br-delivers, and because the first part of line 5 imposes a FIFO order delivery, we have $co\_del_j[s] < sn(m_{k+1})$. Including the bound from (2), we obtain:

$$sn(m_{k+1}) > co\_del_j[s] \geq \mathsf{max}\big\{sn(m) \text{ such that } m \in \{m_1, \dots, m_k\} \wedge proc(m) = s\big\}. \quad (4)$$

Rewriting using (3) we obtain:

$$sn(m_{k+1}) > co\_del_j[s] \geq sn(m_{k+1}) - 1. \quad (5)$$

Thus $sn(m_{k+1}) = co\_del_j[s]+1$ which is the first condition of line 5, thus the wait statement at line 5 terminates at this moment and p_j bco-delivers m_{k+1}.

$\Box_{Theorem\ 2}$

Theorem 3. *Algorithm 1 satisfies the* BCO-Fifo-order *and* BCO-Local-order *properties.*

Proof

- Proof of the BCO-Fifo property. Suppose a correct process p_i bco-delivers first a message m, then a message m', both from the same sender p_j. These messages were br-delivered to p_i with some sequence numbers, sn and sn'. As p_i bco-delivers message from any process in the order defined by their sequence numbers, we have $sn < sn'$. It follows then from the MBR-Termination-2 property that all correct processes bco-deliver m and m' with sequence numbers sn and sn', respectively. It follows that any correct process bco-delivers m before m'.

  If the sender p_j of m and m' is correct, it associated the sequence number sn with m and the sequence number $sn' > sn$ with m'. It follows that p_j bco-broadcast m before m'.

- Proof of the BCO-Local-order property. Let $\rightarrow$ the relation defined on application messages as follows:

R1 If m is a message bco-delivered by a correct process p_i, and $(\langle i, sn\rangle, \langle cb(m), m\rangle)$ the associated protocol message, then for each $\langle j, sn'\rangle \in cb(m)$ we have $msg(j, sn') \to m$.

R2 If m and m' are any two messages successively bco-broadcast by the same correct process p_i, with no other bco-broadcast invocation between them, then we have $m \to m'$.

The condition in the wait() statement at line 5 directly encodes the relation $\to$. Therefore, if $m \to m'$, no correct process bco-delivers m' before m. Let $\to^*$ denote the reflexo-transitive closure of $\to$. The bco-delivery order at correct processes also respects $\to^*$.

Let p_i be a correct process that bco-delivers a message m before bco-broadcasting a message m', and p_j a correct process that bco-delivers m'. We have to show that p_j bco-delivers m before m'.

To this end, let m'' be the first message that p_i bco-broadcast after bco-delivering m. We have $m'' \to^* m'$ (either because $m'' = m'$ or by recursively applying (R2)). What remains to be shown is that $m \to^* m''$.

Let us now consider the sequence of messages $m_1, m_2, \ldots, m_z$ bco-delivered by p_i, starting at $m_1 = m$ and ending at m_z, which is the last message p_i bco-delivered before bco-broadcasting m''. Let cb_i^k be the value of cb_i just before p_i bco-delivered m_k (lines 6-7). To prove BCO-Local Order we need to show that

$$\forall\, k \in \{1, \ldots, z\} :\ \exists\, m'_k \text{ such that } id(m'_k) \in cb_i^k \ \wedge\ m \to^* m'_k.$$

For $k = 1$, due to the update of cb_i just before the bco-delivery of m_1, we have $id(m_1) \in cb_i^1$. As $m_1 \to^* m_1$, the property follows. Let us now assume that, for a given $k < z$ we have $\exists\, m'_k$ such that $id(m'_k) \in cb_i^k \ \wedge\ m \to^* m'_k$. When p_i bco-delivers m_{k+1} there are two cases.

- If $id(m'_k) \notin cb(m_{k+1})$, due to update of cb_i just before the bco-delivery of m_{k+1} at p_i (line 6), we have $id(m'_k) \in cb_i^{k+1}$. We then take $m'_{k+1} = m'_k$.
- If $id(m'_k) \in cb(m_{k+1})$, due to (R1) we have $m'_k \to m_{k+1}$. Moreover, due to the update of cb_i at line 5, we have $id(m_{k+1}) \in cb_i^{k+1}$. Hence m_{k+1} is such that $id(m_{k+1}) \in cb_i^{k+1} \ \wedge\ m \to^* m'_{k+1}$. We then take $m'_{k+1} = m_{k+1}$.

Therefore, in both case we have $\exists\, m'_{k+1}$ such that $id(m'_{k+1}) \in cb_i^{k+1} \ \wedge\ m \to^* m'_{k+1}$. It follows by induction that $\exists\, m'_z$ such that $id(m'_z) \in cb_i^z \ \wedge\ m \to^* m'_z$.

Due to its definition, m_z is the last message bco-delivered by p_i before bco-broadcasting m'', thus $cb(m'') = cb_i^z$. Moreover, we have $id(m_z) \in cb(m'')$, therefore due to (R1) we have $m_z \to m''$. It follows that $m \to^* m''$, from which we conclude that p_j bco-delivers m before m', which concludes the proof of the theorem.

$\square_{Theorem\ 3}$

Theorem 4. *Algorithm 1 satisfies the* BCO-Causality *property, thus it implements the BCO-broadcast abstraction.*

Proof The proof is an immediate consequence of Theorem 1 applied to the results of Theorem 2 and Theorem 3. $\square_{Theorem\ 4}$

7 Byzantine Causal Broadcast in Action

7.1 Building the partial order on the messages

If needed by the application, each correct process p_i can easily build the partial order on the set of messages it bco-delivers. Let $poset_i$ be the message causality directed graph (initially empty) known by process p_i. A vertex is a message identity $\langle j, sn \rangle$. It is initially not marked, and becomes marked when the corresponding message is bco-delivered.

To build the graph $poset_i$, we modify Algorithm 1 as described in Algorithm 2. Lines 1-4 are the same. But, for clarity, we use a different invocation model for bco_deliver() than in Algorithm 1. When the argument of the wait operation in line 5 is satisfied, Algorithm 2 marks the message as "ready to be delivered" by adding a new vertex and its associated directed edges to the graph $poset_i$ (line 7-M, replacing line 7). The application then chooses to deliver messages when needed by invoking bco_deliver() (lines 9-12) explicitly as opposed to using a callback like in Algorithm 1.

```
when $(\langle j, sn \rangle, \langle cb(m), m \rangle)$ is br_delivered from $p_j$
(5)      wait$\big((sn = co\_del_i[j] + 1) \wedge (\forall \langle k, sn' \rangle \in cb(m) :\ co\_del_i[k] \geq sn')\big)$
                $\wedge$(validity_pred$(j, m)$)$)$;   % application-defined predicate
(6)      $cb_i \leftarrow (cb_i \setminus cb(m)) \cup \{\langle j, sn \rangle\}$;
(7-M)    add to the causality graph $poset_i$ the vertex $\langle j, sn \rangle$
           and a directed edge pointing to it from each vertex in $cb_i$
           and a directed edge from the vertex $\langle j, sn-1 \rangle$;
(8)      $co\_del_i[j] \leftarrow co\_del_i[j] + 1$.

operation bco_deliver() at $p_i$ is
(9)      wait($poset_i$ has a non-marked vertex);
(10)     let $\langle k, sn \rangle$ be a minimal non-marked vertex;
(11)     mark the vertex $\langle k, sn \rangle$;
(12)     return( the message identified $\langle k, sn \rangle$).
```

Algorithm 2: Extended algorithm (code for p_i)

7.2 Money transfer based on BCO-broadcast

A money transfer algorithm Guerraoui et al. presented in [22] an important result related to money transfer, a service that is for instance provided by cryptocurrencies [37]. One of the main issues in such systems is to prevent *double spending attacks*, i.e. to prevent users from using the very same money to buy different goods. In [22], Guerraoui et al. proposed two algorithms to solve the money transfer problem when, for each account, there is a single process that is allowed to transfer money from this account to other accounts. This process is called the *owner* of the account[7].

The first algorithm, which only tolerates crash failures, relies on a single writer/multi reader *snapshot object* [1], which can be implemented despite asynchrony and process failures[8]. The second algorithm is made to tolerate Byzantine failures. It eschews snapshot objects and directly implements money transfers in an asynchronous message-passing system where up to $t < n/3$ processes can be Byzantine. This algorithm, which uses an underlying secure broadcast abstraction [33], requires processes to exchange message carrying "histories" which are data structures that grow indefinitely.

A rewriting of Guerraoui et al.'s money transfer algorithm The reader is referred to [22] for a formal specification of the money transfer problem and the proof of the associated algorithms. Algorithm 3, described below, is a simple rewriting based on BCO-broadcast of the message-passing algorithm presented and proved in [22]. A process invokes the operation read(j) to know the balance of p_j, and the operation transfer(j) to transfer the amount v to p_k. Without loss of generality, we assume a correct process p_i does not transfer money to itself.

[7]Note that the notion of *account ownership* here is not the same as in the case of cryptocurrencies, where ownership of an account is defined by the knowledge of a cryptographic secret key that allows one to sign money transfer operations from the account. Here, account owners must be processes in the system.

[8]Namely, the consensus number of a snapshot object is 1 [24, 40].

As in [22], a money transfer of the quantity v by a process p_j to a process p_k is internally represented by a pair $\langle k, v\rangle$, and each process p_i manages a local multiset[9] denoted $hist_i[1..n]$, which records all the transfer operations as known by p_i. More explicitly, $\langle k, v\rangle \in hist_i[j]$ means that p_i knows that p_j transferred the quantity v to p_k. In particular, $hist_i[i]$ contains all the transfers performed by p_i. In our formulation, two transfers of the same amount v by the same process p_j to the same process p_k, appear as identical pairs ($\langle k, v\rangle$) in the multiset $hist_i[j]$ of a process p_i. This multi-set representation is equivalent to that of [22], in which $hist_i[j]$ is a set that stores quadruples of the form $\langle j, sn, k, v\rangle$ instead of pairs: in the quadruple, sn is the sequence number associated by p_j with the transfer identified by $\langle k, v\rangle$.[10]

```
init $init[1..n]$: constant array where $init[k]$ is the initial value of $p_k$ account;
     $hist_i[1..n] \leftarrow [\emptyset, \cdots, \emptyset]$.

function balance($j$) is
(1)  $plus \leftarrow$ sum of the $v_x$ such that $\langle j, v_x\rangle \in \cup hist_i[\ell]$;
(2)  $minus \leftarrow$ sum of the $v_x$ such that $\langle -, v_x\rangle \in hist_i[j]$;
(3)  return($init[j] + plus - minus$).

operation read($j$) is
(4)  return(balance($j$)).

operation transfer($j, v$) is
(5)  if (balance($i$) $< v$)
(6)     then return(abort)
(7)     else $done_i \leftarrow$ false; bco_broadcast TRANSFER($\langle j, v\rangle$);
(8)          wait ($done_i$); return(commit)
(9)  end if.

when TRANSFER($\langle k, v\rangle$) is bco_delivered from $p_j$ do
% The occurrence of this event is immediately followed
% by the atomic execution by $p_i$ of the next two lines
(10) $hist_i[j] \leftarrow hist_i[j] \cup \{\langle k, v\rangle\}$;
(11) if ($j = i$) then $done_i \leftarrow$ true end if

predicate validity_pred($j$,TRANSFER($\langle k, v\rangle$)) is
(12) balance($j$) $\geq v$.
```

Algorithm 3: BCO-broadcast-based rewriting of [22] (code for p_i)

BCO-broadcast-based description of the algorithm Let us first look at the function balance(j) invoked by p_i. All the processes are initialized with the same constant array $init[1..n]$ where $init[k]$ is the initial value of p_k's account. The transfers from p_ℓ to the other processes that are known to p_i are stored in $hist_i[\ell]$, as explained just above. Hence, from p_i's local point of view, $plus$ collects all the transfers to p_j, (line 1), while $minus$ collects all the transfers from p_j (at line 2). When invoked by a process p_i, the operation read(j) returns the current value of p_j's account, as known by p_i (line 3).

When invoked by a process p_i, the operation transfer(j, v) transfers the quantity v from p_i to p_j. To this end, p_i first checks its own account balance (line 5). If the balance is not greater than or equal to v, the transfer aborts (line 6). Otherwise, the transfer can occur. In this case, p_i bco-broadcasts the message TRANSFER($\langle j, v\rangle$) (line 7) and waits until the transfer is locally committed (line 8).

[9]Sometimes also called a *bag* or a *pool*, a multiset is a collection of elements in which the same element can appear several times. As an example, while $\{a, b, c\}$ and $\{a, b, a, b, b, c\}$ are the same set, they are different multisets.

[10]In both algorithms described in [22] and in the rewriting we present, additional bookkeeping information can be stored in $hist_i[j]$, according to application requirements.

When p_i bco-delivers a message $\text{TRANSFER}(\langle k, v\rangle)$ issued by a process p_j, it locally updates its view concerning p_j's transfers, namely $hist_i[j]$ (line 10). Moreover, if p_j is p_i, it switches $done_i$ to `true` (line 11), which allows it to terminate its transfer (line 8).

To prevent over-spending, the predicate $\mathsf{validity\_pred}(j, m)$ where $m = \text{TRANSFER}(\langle k, v\rangle)$ must guarantee that each correct process p_i locally sees that p_j owns enough money in its account to transfer the quantity v to p_k. Hence $\mathsf{validity\_pred}(j, \text{TRANSFER}(\langle k, v\rangle)) \overset{\text{def}}{=} \mathsf{balance}(j) \geq v$[11] (line 12). Since a money transfer that has been accepted can never be undone, double-spending attacks cannot occur in this system. It is not difficult to see that the validity predicate of Algorithm 3 fully maintains the termination of BCO-broadcast, as: (i) a correct process p_j ensure that $\mathsf{balance}(j) \geq v$ before broadcasting $\text{TRANSFER}(\langle k, v\rangle)$, and (ii) $\mathsf{balance}(j) \geq v$ can only be invalidated by withdrawing money from p_j's account, which only p_j can do.

8 Conclusion

After proposing a definition of the Byzantine causal broadcast abstraction, this paper presented an algorithm that implements this specification in asynchronous message-passing systems (a simplified version of it can be trivially obtained if one is interested in FIFO message delivery only). As far as we know, this algorithm is the first ensuring causal message delivery in the presence of Byzantine processes.

Its design relies on a modular decomposition (such as the ones presented in [23, 39]), which makes algorithms easier to understand and prove. In particular, the algorithm relies on an underlying multi-shot reliable broadcast algorithm. It works with any such underlying algorithm and inherits is computability bound and time/message complexities. When instantiated with the reliable broadcast algorithm described in [7] it requires $t < n/3$ (which is resilience optimal), 3 consecutive communication steps and $(n-1)(2n+1)$ protocol messages. When instantiated with the reliable broadcast algorithm described in [26] it requires $t < n/5$, 2 consecutive communication steps (which is optimal), and $n^2 - 1$ protocol messages.

Acknowledgments

The authors want to thank the reviewers for their constructive comments which helped improve both the content and the presentation of the article. This work was partially supported by the French ANR project 16-CE40-0023-03 DESCARTES devoted to layered and modular structures in distributed computing, by the "Web Of Browsers" project granted by the Labex CominLabs excellence laboratory (ANR-10-LABX-07-01), and by the Region Brittany, under the funding program "Stratégie d'Attractivité Durable".

References

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M., and Shavit N., Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890 (1993)

[2] Attiya H., Ellen F., and Morrison A., Limitations of highly-available eventually-consistent data stores. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):141–155 (2017)

[3] 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)

[4] 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)

[11] As the reader can see, $\mathsf{validity\_pred}(j, \text{TRANSFER}(\langle k, v\rangle))$ ensures that the bco-delievery of a transfer message from p_j to p_k and the check that p_j has enough money appear as being done in a single atomic step.

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

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

[7] Bracha G., Asynchronous Byzantine agreement protocols. *Information & Computation*, 75(2):130-143 (1987)

[8] Bracha G. and Toueg S., Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824-840 (1985)

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

[10] Cachin Ch., Kursawe K., Petzold F., and Shoup V., Secure anf efficient asynchronous broadcast protocols. *Proc. 21st Annual Int'l Cryptology Conference (CRYPTO'01)*, Springer LNCS 2139, pp. 524-541 (2001)

[11] Clark D.D., The structuring of systems using upcalls. *Proc. 10th ACM Symposium on Operating Systems Principles (SOSP'85)*, ACM Press, pp. 171-180 (1985)

[12] Coulouris G., Dollimore J., Kindberg K., and Blair G., *Distributed systems: concepts and design, 5th Edition*, Addison Wesley/Pearson Education, ISBN 0-13-214301-1 (2011)

[13] Correia M., Ferreira Neves N., and Veríssimo P., From consensus to atomic broadcast: time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82-96 (2006)

[14] Crain T., Gramoli V., Larrea M., and Raynal M., DBFT: Efficient Byzantine consensus with a weak coordinator and its application to consortium blockchains. *17th IEEE International Symposium on Network Computing and Applications (NCA'18)*, IEEE Press, 8 pages (2018)

[15] De Candia G., Hastorun D., Jampani M., Kakulapati G., Lakshman A., Pilchin A., Sivasubramani S., Vosshall S., and Vogels W., Dynamo: Amazon's highly available key-value store. *SIGOPS Operating Systems Review*, 41(6):205–220 (2007)

[16] Duan S., Reiter M.K., and Zhang H., Secure causal atomic broadcast revisited. *Proc. 47th Int'l conference on Dependable Systemd and Networks (DSN'17)*, IEEE Press, pp. 61-72 (2017)

[17] Fidge C. J., Timestamps in Message-Passing Systems That Preserve the Partial Ordering *Proc. of the 11th Australian Computer Science Conference (ACSC'88)*, pp. 56-66 (1988)

[18] Fox A. and Brewer R., Harvest, yield and scalable tolerant systems. *Proc. 7th Workshop Hot Topics in Operating Systems (HotOS 99)*, IEEE Computer Press, pp. 174–178 (1999)

[19] Fowler M, Module asssembly. *IEEE Software*, 21(2):65–67 (2004)

[20] Gilbert S. and Lynch N., Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59 (2002)

[21] Gilbert S. and Lynch N., Perspectives on the CAP theorem. *Computer*, 45(2):30–36 (2012)

[22] Guerraoui R., Kuznetsov P., Monti M., Pavlovič M., and Seredinschi D.-A., The consensus number of a cryptocurrency. *Proc. 38th ACM Symposium on Principles of Distributed Computing (PODC'19)*, ACM press, pp. 307-316 (2019)

[23] Hadzilacos V. and Toueg S., A modular approach to fault-tolerant broadcasts and related problems. *Tech Report 94-1425*, 83 pages, Cornell University, Ithaca (1994)

[24] Herlihy M.P., Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149 (1991)

[25] Herlihy M.P., Kozlov D., and Rajsbaum S., *Distributed computing through combinatorial topology*, Morgan Kaufmann/Elsevier, 336 pages, ISBN 9780124045781 (2014)

[26] Imbs D. and Raynal M., Trading t-resilience for efficiency in asynchronous Byzantine reliable broadcast. *Parallel Processing Letters*, Vol. 26(4), 8 pages (2016)

[27] 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)

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

[29] Lamport L., Shostack R., and Pease M., The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3)-382-401 (1982)

[30] Lesani M., Bell C.-J., and Chlipala A., Chapar: Certified causally consistent distributed key-value stores. *SIGPLAN Notices*, 51(1):357–370 (2016)

[31] Lloyd W., Michael J. Freedman M. J., Kaminsky M., and Andersen D.-G.,. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, ACM Press, pp. 401–416, (2011)

[32] Lynch N. A., *Distributed algorithms*. Morgan Kaufmann Pub., San Francisco (CA), 872 pages, ISBN 1-55860-384-4 (1996)

[33] Malkhi D. and Reiter M.K., A high throughput secure reliable multicast protocol. *Journal of Computer Security*, 5(2):113-128 (1997)

[34] Mattern F., Virtual Time and Global States of Distributed Systems. *Proc. Workshop on Parallel and Distributed Algorithms*, Elsevier, pp. 215-226, (1988)

[35] Mostéfaoui A., Perrin M., Raynal M., and Cao J., Crash-tolerant causal broadcast in $O(n)$ messages. *Information Processing Letters*, Vol. 151, 6 pages, https://doi.org/10.1016/j.ipl.2019.105837 (2019)

[36] 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)

[37] Nakamoto S., *Bitcoin: A peer-to-peer electronic cash system*, `https://bitcoin.org/bitcoin.pdf` (2008), last accessed March 31, 2020

[38] Oster G., Urso P., Molli P., and Imine A., Data consistency for p2p collaborative editing. *Proc. 20th ACM Conference on Computer Supported Cooperative Work (CSCW'06)*, ACM Press, pp. 259–268 (2006)

[39] Raynal M., *Fault-tolerant message-passing distributed systems: an algorithmic approach.* Springer, 480 pages, ISBN 978-3-319-94140-0 (2018)

[40] Raynal M., An Informal visit to the wonderful land of consensus numbers and beyond. *Bulletin of the European Association of Theoretical Computer Science (EATCS)*, 129:168-192 (2019)

[41] 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)

[42] Reiter M.K. and Birman K.P., How tp securely duplicate services. *ACM Transactions on Programming Languages and Systems.* 16(3):9861009 (1994)

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

[44] Shapiro M., Preguiça N., Baquero C., and Zawirski M., Conflict-free replicated data types. *Proc. 13th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'11)*, Springer LNCS 6976, pp. 386-400 (2011)

[45] Schmuck F., *The use of efficient broadcast in asynchronous distributed systems*. Doctoral Dissertation, Tech. Report TR88-928, Dept of Computer Science, Cornell University, 124 pages (1988)

A Two Signature-free Multi-shot BR-broadcast Algorithms

In order to make the paper as self-contained as possible, this section presents multi-shot extensions of the one-shot signature-free BR-broadcast algorithms introduced by G. Bracha [7] and D. Imbs and M. Raynal [26]. The presentation follows pages 64-71 of [39], where the reader can also find proofs of these algorithms. In the text of these two extensions, sn denotes the sequence number of the corresponding BR-broadcast instance, hence a process invokes br_broadcast(sn, m).

A.1 Underlying basic communication system

Both the algorithms described below, the processes communicate by exchanging messages through an asynchronous reliable point-to-point network. "Asynchronous" means that a message that has been sent is eventually received by its destination process, i.e., there is no bound on message transfer delays. "Reliable" means that the network does not loose, duplicate, modify, or create messages. "Point-to-point" means that there is a bi-directional communication channel between each pair of processes.

A process p_i sends a message to a process p_j by invoking the primitive "send TAG(m) to p_j", where TAG is the type of the message and m its content. To simplify the presentation, it is assumed that a process can send messages to itself. A process receives a message by executing the primitive "receive()". The macro-operation "broadcast TAG(m)" is a shortcut for "**for** $j \in \{1, \cdots, n\}$ **do** send TAG(m) to p_j **end for**".

A.2 Multi-shot version of Bracha's BR-broadcast algorithm

This algorithm assumes $t < n/3$. When, on its client side, a process p_i invokes br_broadcast(sn, m), it invoke the macro-operation broadcast() with the protocol message INIT(sn, m) (line 1).

```
operation br_broadcast($sn, m$) is
(1)   broadcast INIT($sn, m$).

when a message INIT($sn, m$) is received from $p_j$ do
(2)   discard the message if it is not the first message INIT($sn, -$) from $p_j$;
(3)   broadcast ECHO($\langle j, sn\rangle, m$).

when a message ECHO($\langle j, sn\rangle, m$) is received from any process do
(4)   if  (ECHO($\langle j, sn\rangle, m$) received from strictly more than $\frac{n+t}{2}$ different processes)
          $\wedge$(READY($\langle j, sn\rangle, m$) not yet broadcast)
(5)     then broadcast READY($\langle j, sn\rangle, m$)
(6)   end if.

when a message READY($\langle j, sn\rangle, m$) is received from any process do
(7)   if  (READY($\langle j, sn\rangle, m$) received from at least $(t+1)$ different processes)
          $\wedge$(READY($\langle j, sn\rangle, m$) not yet broadcast)
(8)     then broadcast READY($\langle j, sn\rangle, m$)
(9)   end if;
(10)  if  (READY($\langle j, sn\rangle, m$) received from at least $(2t+1)$ different processes)
          $\wedge$ ($\langle j, sn\rangle, m$) not yet br_delivered from $p_j$)
(11)    then br_delivery of $(sn, m)$ from $p_j$
(12)  end if.
```

Algorithm 4: Multi-shot version of Bracha's BR-broadcast algorithm ($t < n/3$, code for p_i)

On it server side a process p_i may receive three different types of protocol messages: INIT(), ECHO(), and READY(). A message INIT carries an application message, while the messages ECHO() and READY()

carry a process identity and an application message[12].

- When p_i receives INIT(sn, m) for the first time from a process p_j (line 2), it broadcasts the protocol message ECHO$(\langle j, sn\rangle, m)$ (line 3). If this message is not the first message INIT$(sn, -)$ from p_j, p_i discards it (in this case, p_j is Byzantine).
- When p_i receives the protocol ECHO$(\langle j, sn\rangle, m)$ for any process, it broadcasts the protocol message READY$(\langle j, sn\rangle, m)$ (line 5) if it received ECHO$(\langle j, sn\rangle, m)$ from enough different processes (where "enough" means here more than $\frac{n+t}{2}$), and READY$(\langle j, sn\rangle, m)$ has not yet been broadcast (line 4). This message exchange ensures that no two correct processes will br-deliver different message from p_j with the sequence number sn, but it is still possible that a correct process br-delivers m from p_j while another correct process does not br-deliver a message from p_j. The role of the message READY$(\langle j, sn\rangle, m)$ is to prevent a correct process from blocking on the br-delivery of m.
- When p_i receives READY$(\langle j, sn\rangle, m)$ for any process, it does the following.
  - Process p_i first broadcasts READY$(\langle j, sn\rangle, m)$ (line 8) if (i) not already done and (ii) it received READY$(\langle j, sn\rangle, m)$ from "enough" processes (where "enough" means here $(t + 1)$ processes, which means from at least on correct process, line 7). As previously indicated, this allows other correct processes not to deadlock.
  - Then, if p_i received READY$(\langle j, sn\rangle, m)$ from "enough" processes (where "enough" means here $((2t + 1)$, which means from at least $(t + 1)$ correct processes), it locally br-delivers the pair (sn, m) (from p_j), if not yet already done (lines 10-11).

This algorithm is optimal with respect to t-resilience (namely $t < n/3$). It requires three consecutive communication steps, and $(n - 1) + 2n(n - 1) = 2n^2 - n - 1$ protocol messages. The proof of this algorithm relies on the following properties, which assume $n > 3t$ (see [39] for their proofs):

- $n - t > \frac{n+t}{2}$.
- Any set containing more than $\frac{n+t}{2}$ different processes, contains at least $(t + 1)$ non-faulty processes.
- Any two sets of processes Q_1 and Q_2 of size at least $\lfloor \frac{n+t}{2} \rfloor + 1$ have at least one correct process in their intersection.

A.3 Multi-shot version of Imbs-Raynal's BR-broadcast algorithm

This algorithm assumes $t < n/5$. The code of br_broadcast(sn, m) is the same as in the previous algorithm.

On its server side a process p_i may receive two different types of protocol messages: INIT() and WITNESS(). The processing of INIT(sn, m) is similar to the one of Algorithm 4. Process p_i simply broadcasts the message WITNESS$(\langle j, sn\rangle, m)$ if it is the first time it received from p_j a message INIT$(sn, -)$ (line 3). Then, when it receives a message WITNESS$(\langle j, sn\rangle, m)$ p_i does the following.

- If it received the same message WITNESS$(\langle j, sn\rangle, m)$ from "enough" processes (where "enough" means here $n - 2t$), and it has not yet broadcast this message (line 4), it does it (line 5).
- If it received WITNESS$(\langle j, sn\rangle, m)$ from "more" processes (where "more" means here $n - t$), and it has not yet br-delivered the pair (sn, m) from p_j (line 7), it br-delivers it (line 8).

[12]The fact that the ECHO() and READY() messages carry a process identity makes redundant the use of an identity in the pair $\langle -, sn\rangle$ that appear in the messages that are br-broadcast. (Hence Algorithm 1 can be modified accordingly.)

```
operation br_broadcast$(sn, m)$ is
(1)   broadcast INIT$(sn, m)$.

when INIT$(sn, m)$ is received from $p_j$ do
(2)   discard the message if it is not the first message INIT$(sn, -)$ from $p_j$;
(3)   broadcast WITNESS$(\langle i, sn\rangle, m)$.

when WITNESS$(\langle j, sn\rangle, m)$ is received from any process do
(4)   if  (WITNESS$(\langle j, sn\rangle, m)$ received from $(n-2t)$ different processes
         $\wedge$ WITNESS$(\langle i, sn\rangle, m)$ not yet broadcast)
(5)     then broadcast WITNESS$(\langle j, sn\rangle, m)$
(6)   end if;
(7)   if  (WITNESS$(\langle j, sn\rangle, m)$ received from $(n-t)$ different processes
         $\wedge (sn, m)$ not yet br_delivered from $p_j$)
(8)     then br_delivery of $(sn, j)$ from $p_j$
(9)   end if.
```

Algorithm 5: Multi-shot version of Imbs-Raynal's BR-broadcast algorithm ($t < n/5$, code for p_i)

Let us notice that, as $t < n/5$, we have $n-2t > 3t$, which means that, in this case, (WITNESS(j, m) was broadcast by at least $n-3t \geq 2t+1$ correct processes. Then, if it received WITNESS(j, m) from more different processes, where "more" means" $(n-t)$, p_i locally br-delivers m from p_j.

As we can easily see, this algorithm requires two communication steps and n^2-1 protocol messages. This better efficiency with respect to Bracha's algorithm is obtained at the price of a weaker t-resilience, namely $t < n/5$.