Byzantine-Tolerant Set-Constrained Delivery Broadcast

Alex Auvolat
École Normale Supérieure, Paris, France
Univ Rennes, Inria, CNRS, IRISA, Rennes, France
https://team.inria.fr/wide/team/alex-auvolat-bernstein/
alex.auvolat@inria.fr

Michel Raynal
Univ Rennes, Inria, CNRS, IRISA, Rennes, France
Department of Computing, Polytechnic University, Hong Kong
https://www.irisa.fr/prive/raynal/
michel.raynal@irisa.fr

François Taïani
Univ Rennes, Inria, CNRS, IRISA, Rennes, France
https://ftaiani.ouvaton.org/
francois.taiani@irisa.fr

Abstract

Set-Constrained Delivery Broadcast (SCD-broadcast), recently introduced at ICDCN 2018, is a high-level communication abstraction that captures ordering properties not between individual messages but between sets of messages. More precisely, it allows processes to broadcast messages and deliver sets of messages, under the constraint that if a process delivers a set containing a message m before a set containing a message m', then no other process delivers first a set containing m' and later a set containing m. It has been shown that SCD-broadcast and read/write registers are computationally equivalent, and an algorithm implementing SCD-broadcast is known in the context of asynchronous message passing systems prone to crash failures.

This paper introduces a Byzantine-tolerant SCD-broadcast algorithm, which we call BSCD-broadcast. Our proposed algorithm assumes an underlying basic Byzantine-tolerant reliable broadcast abstraction. We first introduce an intermediary communication primitive, Byzantine FIFO broadcast (BFIFO-broadcast), which we then use as a primitive in our final BSCD-broadcast algorithm. Unlike the original SCD-broadcast algorithm that is tolerant to up to $t < n/2$ crashing processes, and unlike the underlying Byzantine reliable broadcast primitive that is tolerant to up to $t < n/3$ Byzantine processes, our BSCD-broadcast algorithm is tolerant to up to $t < n/4$ Byzantine processes. As an illustration of the high abstraction power provided by the BSCD-broadcast primitive, we show that it can be used to implement a Byzantine-tolerant read/write snapshot object in an extremely simple way.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Software and its engineering → Abstraction, modeling and modularity

Keywords and phrases Algorithm, Asynchronous system, Byzantine process, Communication abstraction, Distributed computing, Distributed software engineering, Fault-tolerance, Message-passing, Modularity, Read/write snapshot object, Reliable broadcast, Set-constrained message delivery

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.6

Acknowledgements This work has been partially supported by the French ANR projects DESCARTES n. ANR-16-CE40-0023, and PAMELA n. ANR-16-CE23-0016.

1 Introduction

Reliable broadcast in asynchronous crash-prone systems

Reliable broadcast is a communication abstraction central to fault-tolerant asynchronous distributed systems. It allows each process to broadcast, with well-defined delivery guarantees,

messages to all processes in the presence of failures. More precisely, it guarantees that non-faulty processes deliver the same set of messages M, including all the messages they broadcast, and a subset of the messages broadcast by faulty processes before they crashed. The fundamental property of the reliable broadcast abstraction asserts that no two non-faulty processes deliver different sets of messages, and a faulty process delivers a subset of the messages delivered by the non-faulty processes (two distinct faulty processes possibly delivering 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. A process commits a Byzantine failure if it behaves arbitrarily (i.e., its behavior is not the one described by the algorithm is it assumed to execute) [10, 12]. Such a failure can be intentional (also called malicious) or the result of a transient fault which altered its intended behavior in an unpredictable way. Bracha [3] introduced an elegant signature-free Byzantine fault-tolerant algorithm for the reliable broadcast abstraction in n-process message-passing asynchronous systems, where up to $t < n/3$ processes may be Byzantine.

Set-Constrained Delivery broadcast in asynchronous crash-prone systems

Set-Constrained Delivery broadcast (SCD-broadcast) was introduced in [8] in the context of crash failures. Rather than individual messages, a process delivers *non-empty sets* of messages, satisfying the following ordering property: if a non-faulty process delivers a set of message $ms1$ containing a message m and later delivers a set of message $ms1'$ containing a message m', no non-faulty process delivers first a set of message $ms2$ containing m' and later a set of message $ms2'$ containing m. This communication abstraction is particularly efficient to build read/write implementable objects, such as the ones described in [14]. It is shown in [8] that, in asynchronous message-passing systems with up to t crashed processes, (a) $t < n/2$ is necessary and sufficient to build SCD-broadcast, and (b) atomic R/W registers and SCD-broadcast have the same computability power.

Content of the paper

In this paper, we introduce a new communication abstraction, which we call BSCD-broadcast, that provides similar guarantees to SCD broadcast in the context of Byzantine failures. The specification and implementation of such a high-level abstraction has a direct practical interest when realizing many replicated objects. As an example, we show how a Byzantine-tolerant single-writer/multi-reader snapshot object [1] can be easily built on top of BSCD-broadcast. More generally, we believe a better understanding of how high-level broadcast abstractions can be implemented in a Byzantine context can help developers design novel and richer zero-trust applications, extending their use and applicability beyond the highly-publicized examples of cryptocurrencies [5, 11] and smart contract platforms [4].

We begin by defining a first abstraction (Section 3.2), BFIFO-broadcast, that guarantees that non-faulty processes deliver messages from each process in their sending order. Since such a sending order cannot be defined for Byzantine processes, the only guarantee provided in that case is that all non-faulty processes deliver the messages from a given Byzantine process in the same order. Depending on the BR-broadcast algorithm used (e.g., [3, 9]), we obtain for BFIFO-broadcast either an algorithm which requires (i) $t < n/3$ and three sequential communication steps, or (ii) $t < n/5$ and two sequential communication steps.

This simple abstraction is then used to build a BSCD-broadcast algorithm (Section 3) under the assumption $t < n/4$. The design of our algorithm differs significantly from the

SCD-broadcast algorithm described in [8], due to the fact that in the crash-failure model, a process behaves correctly until it possibly crashes, whereas a Byzantine process can exhibit an arbitrary behavior at any time. Precise definitions of both of these communication abstractions are provided in the paper. Our BSCD-broadcast algorithm requires two sequential BFIFO-broadcast steps for each BSCD-broadcast message. These two algorithms are built upon a signature-free Byzantine reliable broadcast primitive and are equally signature-free.

Finally, we build a Byzantine-tolerant snapshot object as an example application of BSCD-broadcast (Section 4). Let us notice that it has recently been shown that the snapshot object is instrumental in the implementation of cryptocurrencies [5]. It follows that the simple implementation (presented below) of such an object in message-passing systems where processes can exhibit Byzantine failures can benefit to cryptocurrencies and some other blockchain-based applications.

2 Computation Model

2.1 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 failures

Up to t processes can exhibit a *Byzantine* behavior. 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. It is however assumed that a Byzantine process cannot send an infinite number of messages in a finite time, a necessary hypothesis in our proof of termination. A process that exhibits a Byzantine behavior is also called *faulty*. Otherwise, it is *correct* or *non-faulty*.

2.2 On the communication side

The basic Byzantine reliable broadcast communication abstraction

This abstraction, denoted BR-broadcast, is a one-shot communication abstraction that provides two operations, br_broadcast() and br_deliver(). "One-shot" means that a process executes br_broadcast() at most once, and br_deliver() at most n times (one per possible sender). As in [6, 13], 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-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. Coupled together, the two termination properties further imply that a message br-broadcast 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 correct processes.

As indicated in the introduction, there are signature-free distributed algorithms, which build BR-broadcast on top of asynchronous message-passing systems in which processes may be Byzantine [3, 9].

Terminology

When studying the BR-broadcast abstraction, a message m br-broadcast by a process is called an *application* message. Differently, a message generated by the algorithm implementing the abstraction is called a *protocol* message. Similarly in the rest of this paper when studying other broadcast abstractions, messages handled by that abstraction are referred to as application messages, and messages generated by the algorithm implementing the abstraction (possibly through a lower-level abstraction) are referred to as protocol messages.

2.3 Multi-shot Byzantine reliable broadcast

As already stated, BR-broadcast is a one-shot communication abstraction. But BR-broadcast allows several processes to invoke the operation br_broadcast(), each giving rise to distinct BR-broadcast instances. Two BR-broadcast instances can be easily distinguished by associating with each of them the identity of the process that created it.

It follows that a multi-shot BR-broadcast abstraction, which we call MBR-broadcast, can be very easily obtained by adding a sequence number sn to each BR-broadcast instance. A MBR-broadcast instance is then identified by a pair $\langle i, sn\rangle$, but only sn needs to be provided in an invocation, namely p_i must invoke br_broadcast(sn_i, m), where sn_i is the local integer variable (initialized to 0) used by p_i to generate its sequence numbers. Conversely, when a message is MBR-delivered by a process p_i, the information provided to the upper layer is a triple $\langle j, sn, m\rangle$ and we say that p_i mbr-delivered the message m from p_j with sequence number sn.

The BR-Validity and BR-Integrity properties for this new multi-shot abstraction become:

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

The other properties MBR-Termination-1 and MBR-Termination-2 remain the same as for BR-broadcast. Multi-shot extensions of the single-shot signature-free BR-broadcast algorithms introduced in [3, 9] are presented in Appendix A.

3 Byzantine Set-Constrained Delivery Broadcast

3.1 Definition

Set-Constrained Delivery broadcast (SCD-broadcast) was introduced in [8], in the context of asynchronous systems prone to process crashes (in which it can be built if and only if $t < n/2$).

We consider here its extension to Byzantine process failures, denoted BSCD-broadcast. This communication abstraction provides two operations, bscd_broadcast() and bscd_deliver(). (We say that a process bscd-broadcasts messages and bscd-delivers messages.) The operation bscd_broadcast(m) allows the invoking process to broadcast an application message, while the operation bscd_deliver() returns a non-empty set of messages to the invoking process. BSCD-broadcast is defined by the following properties.

- BSCD-Validity. If a correct process bscd-delivers a set of messages containing a message m from a process p_i, if p_i is correct, it bscd-broadcast m.
- BSCD-Integrity. A message is bscd-delivered at most once by each correct process.
- BSCD-Order. Let p_i be a correct process that first bscd-delivers a set of messages ms_i and later bscd-delivers a set of messages ms'_i. For any pair of messages $m \in ms_i$ and $m' \in ms'_i$, no correct process bscd-delivers first a set containing m' and later bscd-delivers a set containing m.
- BSCD-Termination-1. If a correct process bscd-broadcasts a message m, it bscd-delivers a message set containing m.
- BSCD-Termination-2. If a correct process bscd-delivers a message set containing m, every correct process bscd-delivers a message set containing m.

As a simple example, let m_1, m_2, m_3, m_4, m_5, m_6, m_7 and m_8 be messages that have been bscd-broadcast by different processes. The following message set bscd-deliveries by p_1, p_2 and p_3 respect the definition of BSCD-broadcast:

- at p_1: $\{m_1, m_2\}$, $\{m_3, m_4, m_5\}$, $\{m_6\}$, $\{m_7, m_8\}$.
- at p_2: $\{m_1\}$, $\{m_3, m_2\}$, $\{m_6, m_4, m_5\}$, $\{m_7\}$, $\{m_8\}$.
- at p_3: $\{m_3, m_1, m_2\}$, $\{m_6, m_4, m_5\}$, $\{m_7\}$, $\{m_8\}$.

Differently, due to the deliveries of the message sets including m_2 and m_3, the following message set deliveries by p_1 and p_2 do not satisfy the BSCD-Order property.

- at p_1: $\{m_1, m_2\}$, $\{m_3, m_4, m_5\}$, ...
- at p_2: $\{m_1, m_3\}$, $\{m_2\}$, ...

3.2 A simple sub-protocol to ease presentation: BFIFO Broadcast

In order to simplify the presentation of our BSCD-broadcast algorithm, we first introduce a straightforward multi-shot first-in-first-out Byzantine broadcast primitive (BFIFO-broadcast), that is implemented on top of MBR-broadcast (Section 2.3) by Algorithm 1. Basing our algorithm on this abstraction allows us to assume that all correct processes agree on the order of relevant messages for any possible sender, *even a Byzantine one*. While ordering messages from each possible sender is an important first step, the crucial property of introducing order between messages of different processes is achieved later, in Algorithm 2.

BFIFO-broadcast is defined by the four MBR-broadcast properties (renamed BFIFO-Validity, BFIFO-Integrity, BFIFO-Termination-1, and BFIFO-Termination-2), to which we add a FIFO delivery guarantee, defined as follows:

- BFIFO-Order. If a correct process p_i bfifo-delivers two messages m and m' from the same process p_k in the order first m and then m', no correct process bfifo-delivers m' before m (BFIFO-Order-1). Moreover, if p_k is correct, it bfifo-broadcast m before m' (BFIFO-Order-2).

In practice, each invocation of bfifo_broadcast() by a process p_i is identified by the pair $\langle i, sn \rangle$, where sn is the corresponding sequence number. The bfifo-delivery order at correct processes corresponds to the order of increasing sequence numbers.

The proof that Algorithm *1* implements BFIFO-broadcast is provided in Appendix B.1.

Algorithm 1 BFIFO-broadcast on top of MBR-broadcast (code for p_i).

init $sn_i \leftarrow 0$; $fifo\_del_i \leftarrow [0, \ldots, 0]$.

operation bfifo_broadcast(m) **at** p_i **is**
(1) $sn_i \leftarrow sn_i + 1$;
(2) br_broadcast(sn_i, m).

when $\langle j, sn, m\rangle$ **is** br_delivered **at** p_i **do**
(3) wait$\big(sn = fifo_del_i[j] + 1\big)$;
(4) bfifo_deliver $\langle j, sn, m\rangle$;
(5) $fifo\_del_i[j] \leftarrow fifo\_del_i[j] + 1$.

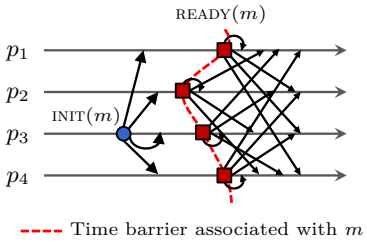

Figure 1 The echo mechanism of Algorithm 2.

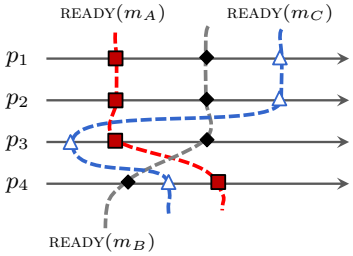

Figure 2 Three time barriers.

3.3 An algorithm for BSCD-broadcast on top of BFIFO-broadcast

Algorithm 2 implements BSCD-broadcast on top of BFIFO-broadcast, with the assumption $t < n/4$. This assumption is required in our proof of correctness (see proof of Lemma 17). An open question remains as to whether $t < n/4$ is a tight bound for BSCD-broadcast.

This algorithm is used in Section 4 to build a Byzantine-tolerant snapshot object.

Overall intuition and challenges

At its core, Algorithm 2 must prevent any potential disruption caused by Byzantine processes, as these cannot be assumed to respect any given behaviour. This need for containment of impredictible behaviour leads to a design that departs fundamentally from the crash-tolerant SCD-broadcast algorithm proposed in [8].

Whereas the crash-tolerant version SCD-broadcast was able to inject order into the system by enforcing a waiting period between successive broadcasts from the same source, this strategy no longer works in a Byzantine setting. This is because nothing prevents Byzantine processes from issuing overlapping broadcasts, in order to confuse correct processes and foil the protocol. This apparently subtle limitation renders the protocol considerably more complex, as correct processes must now cooperate to enforce an order on the broadcasts initiated by a possibly Byzantine source, while pruning inconsistent control information produced by Byzantine processes.

More concretely, Algorithm 2 exploits an echo mechanism (the READY messages bfifo-broadcast at lines 5 and 23, illustrated in Figure 1) to construct temporal barriers that witness a message's distribution among participants. The construction of these barriers is however constrained by preventing correct processes from contributing to the barrier of a bscd-broadcast from a source p_i, if an earlier bscd-broadcast by the same p_i (*earlier* in the sense of the bfifo-broadcast) has not yet been bscd-delivered. This sequencing introduces some order into the set of bscd-broadcast messages unfolding concurrently, and eventually

Algorithm 2 BSCD-broadcast on top of BFIFO-broadcast ($t < n/4$, code for p_i).

init $pending_i \leftarrow [\emptyset, \ldots, \emptyset]$;
  for each $j \in \{1, \cdots, n\}$, $snj \geq 1$
    do $data_i[j, snj] \leftarrow \perp$; $witness_i[j, snj] \leftarrow [+\infty, \cdots, +\infty]$
  end for.

operation bscd_broadcast(m) **at** p_i **is**
(1) bfifo_broadcast INIT(m).

when $\langle j, snj, \text{INIT}(m)\rangle$ **is** bfifo_delivered **at** p_i
(2) % snj is the sequence number carried by INIT(m); namely: $fifo\_del_i[j] = snj$ (cf. Alg. 1)
(3) $data_i[j, snj] \leftarrow m$; % we then have $\mathsf{id}(m) = \langle j, snj\rangle$
(4) $pending_i[j] \leftarrow pending_i[j] \cup \{snj\}$;
(5) **if** $snj = \mathsf{min}(pending_i[j])$ **then** bfifo_broadcast READY($\langle j, snj\rangle$) **end if**;
(6) try_deliver().

when $\langle k, snk, \text{READY}(\langle j, snj\rangle)\rangle$ **is** bfifo_delivered **at** p_i
(7) % snk is the sequence number carried by READY($\langle j, snj\rangle$); namely: $fifo\_del_i[k] = snk$
(8) **if** $witness_i[j, snj][k] = +\infty$ **then** $witness_i[j, snj][k] \leftarrow snk$ **end if**;
(9) **if** $snj > fifo\_del_i[j] \ \wedge \ |\{k' \text{ such that } witness_i[j, snj][k'] < +\infty\}| \geq t + 1$
(10) **then** $pending_i[j] \leftarrow pending_i[j] \cup \{snj\}$ **end if**;
(11) try_deliver().

internal operation try_deliver () **is**
(12) $candidates_i \leftarrow \{\langle j, snj\rangle$ such that
  $(snj \in pending_i[j]) \wedge (|\{k \text{ such that } witness_i[j, snj][k] < +\infty\}| \geq t + 1)\}$;
(13) $todel_i \leftarrow \{\langle j, snj\rangle \in candidates_i$ such that
  $(data_i[j, snj] \neq \perp) \wedge (|\{k \text{ such that } witness_i[j, snj][k] < +\infty\}| \geq n - t)\}$;
(14) **while** $\exists \langle j, snj\rangle \in todel_i : \neg \big(\forall \langle j', snj'\rangle \in candidates_i \setminus todel_i, \mathsf{safe}(\langle j, snj\rangle, \langle j', snj'\rangle)\big)$
(15) **do** $todel_i \leftarrow todel_i \setminus \{\langle j, snj\rangle\}$
(16) **end while**;
(17) **if** $todel_i \neq \emptyset$ **then**
(18) $ms \leftarrow \{\langle j, snj, data_i[j, snj]\rangle$ such that $\langle j, snj\rangle \in todel_i\}$;
(19) bscd_deliver(ms);
(20) **for each** $\langle j, snj\rangle \in todel_i$ **in increasing lexicographical order do**
(21) $pending_i[j] \leftarrow pending_i[j] \setminus \{snj\}$;
(22) **if** $(pending_i[j] \neq \emptyset) \wedge (data_i[j, \mathsf{min}(pending_i[j])] \neq \perp)$
(23) **then** bfifo_broadcast READY($\langle j, \mathsf{min}(pending_i[j])\rangle$) **end if**
(24) **end for**
(25) **end if**.

internal predicate safe ($\langle j, snj\rangle, \langle j', snj'\rangle$) **is**
(26) **if** $(j = j' \wedge snj < snj')$ **then** return (`true`) **end if**;
(27) **if** $|\{k \text{ such that } witness_i[j, snj][k] < witness_i[j', snj'][k]\}| > \frac{n}{2}$ **then** return (`true`) **end if**;
(28) return (`false`).

ensures that received bscd-broadcasts can be locally bundled together and delivered as sets respecting the BSCD-broadcast specification by correct processes.

A note on sequence numbers and message identity

Application messages, each corresponding to an invocation of bscd_broadcast(), are identified by a pair $\langle j, snj\rangle$ made of a process identity j and a sequence number snj. To simplify the presentation of the algorithm, and without loss of generality, the sequence numbers given to bscd-broadcast application messages correspond to the sequence number of an underlying bfifo-broadcast message noted INIT(m) in the algorithm. However the BSCD-broadcast algorithm also bfifo-broadcasts protocol messages of other types (namely, READY messages), therefore many sequence numbers of bfifo-broadcast messages do not correspond to a bscd-broadcast message. As a consequence, these application messages do not have strictly sequential numbers.

Local variables at a process

Each process p_i manages the following local variables:

- $pending_i[1..n]$ is an array of sets, such that $pending_i[j]$ contains the sequence numbers of the messages known by p_i as bscd-broadcast by p_j and not yet locally bscd-delivered.
- $data_i[1..n, 1...]$ is a two-dimensional array, whose entries are initially $\perp$. $data_i[j, snj]$ stores the content of message $\langle j, snj\rangle$ (the message from p_j whose sequence number is snj).
- $witness_i[1..n, 1...]$ is another two-dimensional array that records the temporal barriers observed so far by p_i. Each entry $witness_i[j, snj]$ stores the barrier for message $\langle j, snj\rangle$, an array of size n initialized to $[+\infty, \cdots, +\infty]$. As the algorithm progresses, $witness_i[j, snj][k]$ is updated to contain the logical date snk (a sequence number of p_k) at which p_k witnessed message $\langle j, snj\rangle$ by bfifo-broadcasting the protocol message READY($\langle j, snj\rangle$).
- $candidates_i$ and $todel_i$ are sets containing the identities of message which are potential candidates to belong to the next message set bscd-delivered by p_i; $todel_i$ is a refined subset of $candidates_i$.

Behavior of a process p_i

A process triggers the bscd-broadcast of a message m by simply bfifo-broadcasting the protocol message INIT(m) (line 1, illustrated in Fig. 1).

When a process p_i bfifo-delivers a message INIT(m) from a process p_j, this protocol message has been allocated a sequence number called snj (line 2) by the underlying BFIFO broadcast algorithm (Section 3.2), and consequently $\langle j, snj\rangle$ becomes the identity of the application message m. m is first stored in $data_i[j, snj]$ (line 3), and its sequence number snj is then added to $pending_i[j]$ (line 4), to record that p_i knows that p_j used an INIT message to trigger a bscd-broadcast for the message with sequence number snj, and that p_i must bscd-deliver this message. Finally, if snj is the smallest sequence number in $pending_i[j]$, p_i knows that all earlier messages from p_j have been bscd-delivered, and that $\langle j, snj\rangle$ is the next message it will bscd-deliver from p_j. It informs of it the other processes by bfifo-broadcasting the protocol message READY($\langle j, snj\rangle$) (line 5, illustrated in Fig. 1) in order to witness the message $\langle j, snj\rangle$. This READY message will contribute to the time barrier needed to order $\langle j, snj\rangle$ with respect to other concurrent bscd-broadcast messages (more on this below). p_i then invokes try_deliver() (line 6) to see if it can bscd-deliver a set of messages.

When a process p_i bfifo-delivers a witness message READY($\langle j, snj\rangle$) from p_k regarding a message $\langle j, snj\rangle$, p_i uses the sequence number of the READY message (called snk, line 7) to update $witness_i[j, snj]$, the time barrier of the message $\langle j, snj\rangle$. Concretely, if $witness_i[j, snj][k] = +\infty$, p_i learns that p_k has witnessed the message identified by $\langle j, snj\rangle$ from p_j, and p_i records the time point at which this happened by updating $witness_i[j, snj][k]$ to snk (line 8). Then, if p_i has not yet bfifo-delivered the message $\langle j, snj\rangle$ (first predicate of line 9) and at least $t + 1$ processes (i.e., at least one correct process) have witnessed this message (by bfifo-broadcasting a READY($\langle j, snj\rangle$) message, second predicate of line 9), it knows that p_j bscd-broadcast a message identified by $\langle j, snj\rangle$ and it adds snj to $pending_i[j]$ (line 10). This early update of $pending_i[j]$ (i.e. before p_i receives the corresponding INIT message from p_j) ensures $\langle j, snj\rangle$ will be taken into account in the tests performed by try_deliver() at lines 12-16 to decide which application messages can be safely bscd-delivered (we return to these tests just below). Finally, after these various updates, p_i invokes try_deliver() (line 11) to attempt to bscd-deliver a set of messages.

Internal operation try_deliver () and predicate safe ()

In this operation, p_i first computes the set $candidates_i$ that contains the message identities $\langle j, snj\rangle$ that have been received but not yet delivered ($snj \in pending[j]$), and that have been witnessed by at least one correct process (first and second predicate of line 12). Then, to obtain the reduced set $todel_i$ (line 13) of messages that can actually be delivered, p_i first purges from $candidates_i$ the message identities whose payload has not been received yet or that have been witnessed by less than $(n-t)$ processes (a process p_k witnesses an application message m by bfifo-broadcasting READY($\langle j, snj\rangle$) at line 5 or 23, whose reception entails the update of $witness_i[j, snj]$ by p_i at line 8, see above).

Once this first purge is complete, p_i then removes from $todel_i$ (line 15) all the message identities $\langle j, snj\rangle$ (line 14) that make it "unsafe" with respect to messages in $candidates_i \setminus todel_i$ (the messages that have been "confirmed", but cannot be delivered yet, line 14), where *unsafe* is defined as the negation of the predicate safe() defined at lines 26-28. Thus a process will only bscd-deliver a set of messages $todel_i$ whose messages are all *safe* to bscd-deliver before those remaining in $candidates_i \setminus todel_i$. More precisely a message m identified by $\langle j, snj\rangle$ is considered *safe* by p_i to bscd-deliver before a message m' identified by $\langle j', snj'\rangle$ if (i) both have been bscd-broadcast by the same process with $snj < snj'$ (line 26) or (ii) p_i knows that a majority of processes witnessed m before m' (line 27). Thus a process may *learn* that it is safe to deliver m before m once it has received enough READY messages for m before READY messages from m' by the same senders. Note that this *safety* relationship is not transitive, and can lead to complex entanglements. For instance, Figure 2 shows the time barriers of three messages, i.e. the times at which correct processes and possibly a subset of Byzantine processes have first bfifo-broadcast READY($m_{\{A,B,C\}}$) messages. In this example, a process may learn that m_A is safe to bscd-deliver before m_B, and that m_B is safe to bscd-deliver before m_C, but never that m_A is safe to bscd-deliver before m_C. Thus correct processes must bscd-deliver m_A, m_B and m_C in the same set.

Finally, if $todel_i$ is not empty (line 17), p_i computes from this set of message identities the triplets $\langle j, snj, data_i[j, snj]\rangle$ (line 18), which define the set of messages ms that it can bscd-deliver (line 19). Then, according to the messages it has just bscd-delivered, p_i updates the sets $pending_i[j]$ (lines 20-21). Moreover, if such a set remains non-empty, and p_i bfifo-delivered the protocol message INIT(m) associated with $\mathsf{min}(pending_i[j])$ (line 22), p_i bfifo-broadcasts the protocol message READY($j, \mathsf{min}(pending_i[j])$ to inform the other processes that this is the identity of the next message it intends to bscd-deliver from p_j (line 24).

▶ **Theorem 1.** *Algorithm 2 respects the properties of* BSCD-Validity, BSCD-Integrity, BSCD-Ordering, BSCD-Termination-1, *and* BSCD-Termination-2.

The following section is devoted to proving this result.

3.4 Proof of Correctness

We start by several intermediate definitions and results before the final proof of Theorem 1. The most important results are shown in Theorem 18 (BSCD-Order) and Theorem 22 (BSCD-Termination).

▶ **Definition 2.** *We say a correct process p_i bscd-delivers a message m from p_j with sequence number snj if it bscd-delivers a message set ms that contains the tuple $\langle j, snj, m\rangle$.*

▶ **Theorem 3.** *If a correct process p_i bscd-delivers a message m from a correct process p_j, then p_j bscd-broadcast m.*

Proof. If a correct process p_i bscd-delivers a set containing a message m from p_j, then m has been put in $data_i[j, snj]$ for some snj, meaning p_i bfifo-delivered INIT(m) from p_j. By BFIFO-Validity, if p_j is correct then it bfifo-broadcast INIT(m), meaning it bscd-broadcast m. ◀

▶ **Definition 4.** *We say a message identity $\langle j, snj\rangle$ is valid if all correct processes bfifo-deliver a message* INIT(m) *from p_j at sequence number snj. Otherwise the message identity $\langle j, snj\rangle$ is invalid.*

▶ **Remark 5.** *If m is a message bscd-broadcast or bscd-delivered by a correct process, then we note* $\mathrm{id}(m) = \langle j, snj\rangle$ *its identity which is by definition a valid identity. We will note* $\mathrm{sender}(m) = j$ *and* $\mathrm{sn}(m) = snj$.

▶ **Lemma 6.** *If at some point $snj \in pending_i[j]$ for a correct process p_i, then $\langle j, snj\rangle$ is a valid message identity, i.e. all correct processes will bfifo-deliver a message* INIT(m) *from p_j with sequence number snj.*

Proof. If snj is added to $pending_i[j]$ at line 4 then p_i bfifo-delivered a message INIT(m) from p_j at snj, so by BFIFO-Termination so will all other correct processes.

If snj is added to $pending_i[j]$ at line 10 then p_i bfifo-delivered READY$(\langle j, snj\rangle)$ from at least $t + 1$ processes, one of which at least is correct. Thus there is a correct process that bfifo-broadcast READY$(\langle j, snj\rangle)$, meaning it bfifo-delivered INIT(m) from p_j at snj, so by BFIFO-Termination, all other correct processes also will. ◀

▶ **Theorem 7.** *No correct process bscd-delivers several (identical or different) messages for a given message identity $\langle j, snj\rangle$.*

Proof. A message that is bscd-delivered by a correct process p_i has a unique identity $\langle j, snj\rangle$ guaranteed by the bfifo-broadcast of the corresponding INIT(m) message at p_j. At a correct process p_i, BFIFO-Integrity guarantees that INIT(m) will be delivered only once with sequence number snj from p_j. snj will be added in $pending_i[j]$ at that time or before but cannot be added again later, and will be removed from $pending_i[j]$ only after INIT(m) has been bfifo-delivered, therefore snj can only be removed from $pending_i[j]$ once meaning that m can be bscd-delivered only once. ◀

▶ **Lemma 8.** *If at any correct processes p_i at any time of its execution, and for any p_j, snj, p_k we have $witness_i[j, snj][k] = snk \neq \infty$ then snk is the sequence number of the first* READY$(\langle j, snj\rangle)$ *message bf-delivered by p_i from p_k, which is the same at every correct process, even if p_k is Byzantine.*

Proof. If $witness_i[j, snj][k]$ is set to other than ∞ it is when p_i bfifo-delivers READY$(\langle j, snj\rangle)$ from p_k. If p_i bfifo-delivers READY$(\langle j, snj\rangle)$ from p_k several times, the first time will lead to the execution of $witness_i[j, snj][k] \leftarrow snk$ and the second one won't because we now have $witness_i[j, snj][k] \neq \infty$ (line 8). Correct processes all bfifo-deliver the same messages from p_k in the same order and with the same sequence numbers, therefore they will eventually all update their $witness_i[j, snj][k]$ variable to the same value. ◀

▶ **Convention 9.** *Since only the first* READY$(\langle j, snj\rangle)$ *bfifo-delivered by a correct process p_i from p_k can lead to a change in $witness_i[j, p_j][k]$, we will say that p_i "received"* READY$(\langle j, snj\rangle)$ *from p_k to refer only to the first such* READY *bfifo-delivered. This is important if the sender of the* READY*, p_k, is Byzantine since in that case it may bfifo-broadcast several times the same* READY *message. We will say that p_i received* READY$(\langle j, snj\rangle)$ *before* READY$(\langle j', snj'\rangle)$ *from p_k*

if p_i bfifo-delivered for the first time READY$(\langle j, snj\rangle)$ *from p_k before bfifo-delivering for the first time* READY$(\langle j', snj'\rangle)$ *from p_k. We will also say that a correct process receives* READY$(\langle j, snj\rangle)$ *before* READY$(\langle j', snj'\rangle)$ *from p_k if it bfifo-delivers at least once* READY$(\langle j, snj\rangle)$ *from p_k and never bfifo-delivers* READY$(\langle j', snj'\rangle)$ *from p_k.*

▶ **Remark 10.** *If a correct process receives* READY$(\langle j, snj\rangle)$ *before* READY$(\langle j', snj'\rangle)$ *from a process p_k, then the same happens for all other correct processes: if p_i only bfifo-delivers one or more* READY$(\langle j, snj\rangle)$ *from p_k and no* READY$(\langle j', snj'\rangle)$ *from p_k then all other correct processes will also bfifo-deliver it or them and they will not bfifo-deliver a* READY$(\langle j', snj'\rangle)$ *from p_k. If p_i both bfifo-delivers one or more* READY$(\langle j, snj\rangle)$ *from p_k and one or more* READY$(\langle j', snj'\rangle)$ *from p_k, other correct processes will also bfifo-deliver them, in the same order and with the same sequence numbers.*

▶ **Lemma 11.** *If at any correct process p_i, for any p_j, snj, p'_j, snj' and p_k, we have $witness_i[j, snj][k] < witness_i[j', snj'][k]$ at any time of its execution, then that will always be the case at p_i afterwards.*

Proof. If $witness_i[j, snj][k] < witness_i[j', snj'][k]$ then $witness_i[j, snj][k]$ was set to a non-infinite value which is the sequence number with which p_i bfifo-delivers the first READY$(\langle j, snj\rangle)$ from p_k. The content of $witness_i[j, snj][k]$ will never be changed afterwards, and if $witness_i[j', snj'][k]$ is changed afterwards it will be set to the sequence number with which p_i bfifo-delivers the first READY$(\langle j', snj'\rangle)$ from p_k which is necessarily larger than the sequence number of the first READY$(\langle j, snj\rangle)$. ◀

We now contruct a relation $\to$ on messages to capture the fact that a message m can be safely bscd-delivered before a message m': if $m \to m'$, no correct process delivers m' before m. The relation $\to$ relies on the time barriers of m and m' arising from the witness messages READY. We construct $\to$ incrementally, starting with a simpler preliminary relation $\prec_k$.

▶ **Definition 12.** *If correct processes receive* READY$(\langle j, snj\rangle)$ *before* READY$(\langle j', snj'\rangle)$ *from p_k, we write: $\langle j, snj\rangle \prec_k \langle j', snj'\rangle$.*

▶ **Remark 13.** *If p_i is a correct process, $witness_i[j, snj][k] < witness_i[j', snj'][k]$ can be interpreted as: "p_i knows that $\langle j, snj\rangle \prec_k \langle j', snj'\rangle$" and implies $\langle j, snj\rangle \prec_k \langle j', snj'\rangle$.*

▶ **Definition 14.** *Let us note $m \to m'$ if* (i) *there is a majority P of processes such that $\forall p_k \in P, m \prec_k m'$, or if* (ii) *$m$ and m' are sent by the same process and $\mathrm{sn}(m) < \mathrm{sn}(m')$. Formally:* $m \to m' \;\Leftrightarrow\; \Big(\mathrm{sender}(m) = \mathrm{sender}(m') \wedge \mathrm{sn}(m) < \mathrm{sn}(m')\Big) \vee \left(|\{k : m \prec_k m'\}| > \frac{n}{2}\right)$.

▶ **Lemma 15.** *$m \to m'$ and $m' \to m$ are exclusive: $\forall m, m', \neg(m \to m' \wedge m' \to m)$.*

Proof. If $\mathrm{sender}(m) = \mathrm{sender}(m')$, then suppose w.l.o.g. that $\mathrm{sn}(m) < \mathrm{sn}(m')$. We directly have $m \to m'$ following from the definition. Moreover, if correct processes bfifo-broadcast READY$(\mathrm{id}(m'))$, they will do it after they have already bfifo-broadcast READY$(\mathrm{id}(m))$, thus $|\{k : m' \prec_k m\}| \leq t$, thus we have $\neg(m' \to m)$.

If m and m' are not from the same sender and $m \to m'$ and $m \to m'$ then we have $|\{k : m \prec_k m'\}| > \frac{n}{2}$ and $|\{k : m' \prec_k m\}| > \frac{n}{2}$. There is a process in the intersection of these two majorities for which we have a contradiction. ◀

▶ **Remark 16.** *$\to$ is not a partial order as it is not transitive.*

▶ **Lemma 17.** *If a correct process p_i bscd-delivers m before bscd-delivering m' or bscd-delivers m and bfifo-delivers* INIT(m') *but never bscd-delivers m' then $m \to m'$.*

Proof. When m is bscd-delivered by p_i, it has received $\text{READY}(\text{id}(m))$ from at least $n-t$ processes. Two cases arise:

- Case 1: If p_i has already received an $\text{READY}(\text{id}(m'))$ from no more than t processes, then there are at least $n-2t$ processes for which p_i has received a READY $(\text{id}(m))$ and not yet a $\text{READY}(\text{id}(m'))$, therefore $|\{k : m \prec_k m'\}| \geqslant n-2t$. The hypothesis of our computing model is $t < n/4$, therefore $n-2t > \frac{n}{2}$. We thus have $m \to m'$.
- Case 2: If p_i has already received an $\text{READY}(\text{id}(m'))$ from $t+1$ or more processes when it bscd-delivered m, then $\text{sn}(m') \in \mathit{pending}_i[k]$ (because of line 9), thus $\text{id}(m') \in \mathit{candidates}_i \backslash \mathit{todel}_i$, thus by the exit condition of the while loop, either $\text{sender}(m) = \text{sender}(m') \wedge \text{sn}(m) < \text{sn}(m')$ or $|\{k : \mathit{witness}_i[\text{id}(m)][k] < \mathit{witness}_i[\text{id}(m')][k]\}| > \frac{n}{2}$, in both cases $m \to m'$. ◀

▶ **Theorem 18.** *Let p_i be a correct process that bscd-delivers a set ms_i containing a message m and later bscd-delivers a set ms'_i containing a message m'. No correct process p_j bscd-delivers first a set ms'_j containing m' and later a message set ms_j containing m.*

Proof. By Lemma 17, such a situation would imply $m \to m'$ and $m' \to m$, which we have shown are exclusive (Lemma 15). ◀

▶ **Definition 19.** *We say that a correct process p_i knows that $m \to m'$ if either $|\{k : \mathit{witness}_i[\text{id}(m)][k] < \mathit{witness}_i[\text{id}(m')][k]\}| > \frac{n}{2}$ or $\text{sender}(m) = \text{sender}(m') \wedge \text{sn}(m) < \text{sn}(m')$.*

▶ **Remark 20.** *If a correct process p_i knows that $m \to m'$ then $m \to m'$.*

▶ **Remark 21.** *In the while loop of* try_deliver(), *a correct process removes a message m from todel_i when there is a message m' in $\mathit{candidates}_i \backslash \mathit{todel}_i$ for which it does not know that $m \to m'$. For a correct process to bscd-deliver a set of messages ms, it must know that $m \to m'$ for all $m \in \text{ms}$ and $m' \in \mathit{candidates}_i \backslash \text{ms}$.*

▶ **Theorem 22.** *If a correct process p_i bfifo-delivers $\text{INIT}(m)$ from p_j, it eventually bscd-delivers m from p_j.*

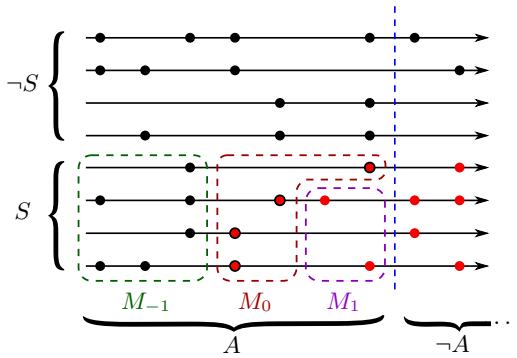

Figure 3 Message sets used in the proof of Theorem 22.

Proof. Suppose by contradiction that there is a correct process that bfifo-delivers an $\text{INIT}(m)$ but never bscd-delivers m. Let M be the set of these *unterminated* messages, i.e. the set of messages m for which correct processes bfifo-deliver $\text{INIT}(m)$ but there is at least one correct process that does not bscd-deliver m.

Let $M_{p_i} = \{m \in M : \text{sender}(m) = p_i\}$, the set of unterminated messages bscd-broadcast by p_i, and $S = \{i : M_{p_i} \neq \varnothing\}$, the set of processes who bscd-broadcast at least one unterminated message. For $i \in S$, let m_i be the message in M_{p_i} with the lowest sequence number, and let $M_0 = \{m_i, i \in S\}$. We use M_0 as a boundary to define four sets of messages that will lead us to a contradiction. Let $M_{-1} = \{m : \text{sender}(m) \in S \wedge \text{sn}(m) < \text{sn}(m_i)\}$ the set of all messages by processes of S that all correct processes bscd-deliver.

After some time, correct processes all bscd-deliver the messages of M_{-1}, thus all correct processes will bfifo-broadcast READY(id(m_i)) for all $i \in S$ after some time. Let A be the set of messages witnessed (with a READY message) by at least one correct process before one of the unterminated messages in the "boundary" M_0. More formally A is defined by: $m' \in A$ iff a correct process bfifo-broadcasts READY(id(m')) before bfifo-broadcasting READY(id(m_i)) for some $i \in S$. Because a process may only receive a finite number of messages in a finite amount of time, the set A is finite. For all $m \notin A$ and $i \in S$ we have $m_i \to m$.

Let $A_S = \{m \in A : \text{sender}(m) \in S\}$ and $A_{\neg S} = \{m \in A : \text{sender}(m) \notin S\}$. Let $M_1 = A_S \backslash (M_{-1} \cup M_0)$. We have that $A = A_{\neg S} \cup M_{-1} \cup M_0 \cup M_1$. We have two cases:

- Case 1: no correct process ever bscd-delivers a message of M_0. In that case, no correct process will ever bfifo-broadcast a READY for a message of M_1, thus the messages of M_1 will never be considered in the $candidates_i$ set of correct processes.

  Let p_i be a correct process. After a certain time p_i will have bscd-delivered all messages of $A_{\neg S}$ and of M_{-1} and will have bfifo-delivered its last READY(id(m)) for a message $m \in M_0$, thus it will execute try_deliver() either strictly after all that happens or when bscd-delivering for the last time messages of $A_{\neg S} \cup M_{-1}$ and after having bfifo-delivered its last READY(id(m)) for a message $m \in M_0$. At that point $candidates_i$ will contain only messages of M_0 and messages not in A, and possibly some messages of $A_{\neg S} \cup M_{-1}$ which will be in $todel_i$ at the end of the while loop. Since p_i never bscd-delivers messages of M_0, at some point in the while loop it will remove a message $m \in M_0$.

  Let m be the first message of M_0 p_i removes from $todel_i$, it is removed because there is an $m' \in candidates_i \backslash M_0$ that violates the exit condition of the loop. We have $m' \notin M_0$ because m is the first message of M_0 that p_i removes from $todel_i$ and initially there is no message of M_0 in $candidates_i \backslash todel_i$, $m' \notin A_{\neg S} \cup M_{-1}$ because all messages of $A_{\neg S} \cup M_{-1}$ remaining at this step are in $todel_i$ at the end of the loop, and $m' \notin M_1$ because no message of M_1 is in $candidates_i$, thus $m' \notin A$. Since $m \in M_0$ and $m' \notin A$ we have $m \to m'$ and since p_i has bfifo-delivered all messages READY(id(m)) then it knows that $m \to m'$, meaning that safe(id(m), id(m')) = true. Therefore p_i cannot remove m from $todel_i$ because of m'. We have a contradiction.

- Case 2: there is a correct process p_i that bscd-delivers a message $m \in M_0$. In that case, p_i delivered a certain message set ms that contains m. Let us note U the set of all messages p_i bscd-delivered before and including ms. For any message m' for which correct processes deliver INIT(m') and such that $m' \notin U$, by Lemma 17 we have $\forall m'' \in U, m'' \to m'$.

  Let p_j be a correct process that does not bscd-deliver m. After a certain time, it will receive its last READY for a message of U (in particular, it will have bfifo-delivered READY(id(m)) from all correct processes), at which time it executes try_deliver() and removes at least one message of U from $todel_j$ (it removes at least m). Moreover no messages of U are in $candidates_i \backslash todel_i$ at the beginning of the loop since p_i was able to bscd-deliver all messages of U and all READY messages for messages of U have been bfifo-delivered by p_j. Let m_0 be the first message of U that p_j removes from $todel_j$. The message m_0 is removed because of some message $m_1 \notin U$ that violated the condition. Since p_i bscd-delivered m_0 before bscd-delivering m_1 we have $m_0 \to m_1$. Since p_j has

already bfifo-delivered its last messages READY(id(m_0)) then it knows that $m_0 \to m_1$, meaning that safe(id(m), id(m')) = `true`. Therefore p_j cannot remove m from $todel_i$ because of m'. We have a contradiction. ◀

Final proof of Theorem 1

Proof. BSCD-Validity, BSCD-Integrity and BSCD-Ordering are shown respectively in Theorem 3, Theorem 7 and Theorem 18. BSCD-Termination-1: if a correct procsess bscd-broadcasts a message m, then it will bf-broadcast INIT(m). By BFIFO-Termination it will thus bf-deliver INIT(m), thus by Theorem 22 it will bscd-deliver m. BSCD-Termination-2: If a correct process bscd-delivers m then it has previously bf-delivered INIT(m). By BFIFO-Termination all correct processes bf-deliver INIT(m). By Theorem 22, all correct processes will bscd-deliver m. ◀

4 BSCD-broadcast in action: a Byzantine-tolerant snapshot object

The snapshot object was introduced in [1, 2]. A snapshot object can be seen as an array $REG[1..n]$ of single-writer/multi-reader atomic registers which provides processes with two operations, denoted write() and snapshot(). The invocation of write(v) by a process p_i assigns atomically v to $REG[i]$. The invocation of snapshot() returns the value of $REG[1..n]$ as if it was executed instantaneously. Hence, in any execution of a snapshot object, its operations write() and snapshot() are linearizable [7]. As mentioned in the introduction, the snapshot object has recently been shown to be instrumental in the design of cryptocurrency systems [5].

Byzantine Snapshot object on top of BSCD-broadcast

Let us recall that nothing can prevent Byzantine processes from writing fake values in their register of the snapshot object. The important property is that the registers associated with correct processes cannot be corrupted by Byzantine processes.

Our algorithm is similar in structure to the multi-writer/multi-reader snapshot object built on SCD-broadcast that was introduced in [8] in the context of process crash failures. However a multi-writer snapshot object is intrinsically unsuited to a system with Byzantine processes as such processes would then be able to prevent all effective communication between correct ones by overwriting correct processes' values as soon as they are written, which is why we focus on a single-writer/multi-reader snapshot object instead. As a consequence, the analysis of our algorithm shares little in common with that of [8].

Local data structures at a process p_i

Let REG be the snapshot object. At a process p_i, it uses three local data structures:

- $done_i$ is a Boolean variable.
- $reg_i[1..n]$ is the value of $REG[1..n]$ as currently known by p_i.
- $wsn_i[1..n]$ is such that $wsn_i[j]$ is the sequence number of the last write by p_j in $REG[j]$, as known by p_i. Such a sequence number is systematically associated (at line 18 of Algorithm 2) with each invocation of bscd_broadcast WRITE() (line 3 of Algorithm 3).

Operation snapshot()

When a process invokes snapshot(), it simply invokes bscd_broadcast SYNC(), where SYNC is a synchronization tag, and (with the help of the Boolean $done_i$, lines 1 and 7) waits until this message has been locally processed. Then, p_i returns the value of the local array reg_i.

Algorithm 3 Construction of a snapshot object on top of BSCD-broadcast (code for p_i).

init $reg_i \leftarrow [\bot, \ldots, \bot]$; $wsn_i \leftarrow [0, \ldots, 0]$.

operation snapshot() **is**
(1) $done_i \leftarrow$ `false`; bscd_broadcast SYNC(); wait($done_i$);
(2) return($reg_i[1..n]$).

operation write(v) **is**
(3) $done_i \leftarrow$ `false`; bscd_broadcast WRITE(v); wait($done_i$).

when the message set $\{ \langle j_1, sn_1, \text{WRITE}(v_1)\rangle, \ldots, \langle j_x, sn_x, \text{WRITE}(v_x)\rangle, \langle j_{x+1}, sn_{x+1}, \text{SYNC}()\rangle, \ldots, \langle j_y, sn_y, \text{SYNC}()\rangle \}$ **is bscd-delivered do**
(4) **for each message** $\langle j, snj, \text{WRITE}(v)\rangle \in$ bscd-delivered message set **do**
(5) **if** $(wsn_i[j] < snj)$ **then** $reg_i[j] \leftarrow v$; $wsn_i[j] \leftarrow snj$ **end if**
(6) **end for**;
(7) **if** $\exists \ell : j_\ell = i$ **then** $done_i \leftarrow$ `true` **end if**.

The aim of the message SYNC is to stop the progress of p_i so that, once unblocked, it will have a consistent value of REG it can return ("consistent" refers here to the atomicity of the snapshot object).

Operation write()

The code of this operation is similar in structure to the synchronization of operation snapshot(), however instead of reading the registers and returning their values, the process sends a value to be written but does not return anything. When a process invokes write(v), it simply invokes bscd_broadcast WRITE(v), where WRITE is an operation tag, and, with the help of the Boolean $done_i$, waits until this message has been locally processed (lines 3 and 7, this synchronization pattern is sometimes called "read your writes").

Processing of a set of messages

This procedure consists in two steps:

- Process p_i first considers the messages WRITE() that appear in the message set ms it is bscd-delivering. Each element of ms is actually a triplet $\langle j, snj, v\rangle$, such that $\langle j, snj\rangle$ is the identity of the value v, namely the snj-th value written by p_j in $REG[j]$. p_i writes v into $reg_i[j]$, if this value has not been overwritten by a more recent value (lines 4-6).
- After the previous updates, p_i sets the Boolean $done_i$ to the value `true` if the message set contains a message it bscd-broadcast (line 7).

▶ **Theorem 23.** *Algorithm* 3 *builds a linearizable* snapshot *object in an n-process asynchronous message-passing system where up to $t < n/4$ processes may commit Byzantine failures.*

The proof of this theorem can be found in Appendix B.2.

5 Conclusion

This paper addressed the design of a Set-Constrained Delivery broadcast abstraction in the context of n-process asynchronous message-passing systems where up to t processes may commit Byzantine failures. A first primitive, BFIFO-broadcast, ensures that, for any sender, all correct processes deliver its messages in the same order (which is their sending order if the sender is correct). BSCD-broadcast, which is built over BFIFO-broadcast, ensures that correct processes deliver sets of messages such that, if a correct process p delivers a

set of messages containing a message m and later delivers a set of messages containing a message m', no correct process delivers first a set of messages containing m' and later a set of messages containing m. As an illustration of BSCD-broadcast, it has been shown how it facilitates the construction of a Byzantine-tolerant read/write snapshot object.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic Snapshots of Shared Memory. *J. ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.

2 James H. Anderson. Multi-Writer Composite Registers. *Distributed Computing*, 7(4):175–195, 1994. doi:10.1007/BF02280833.

3 Gabriel Bracha. Asynchronous Byzantine Agreement Protocols. *Inf. Comput.*, 75(2):130–143, 1987. doi:10.1016/0890-5401(87)90054-X.

4 Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding Concurrency to Smart Contracts. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 303–312. ACM, 2017. doi:10.1145/3087801.3087835.

5 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The Consensus Number of a Cryptocurrency. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019.*, pages 307–316. ACM, 2019. doi:10.1145/3293611.3331589.

6 Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

7 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

8 Damien Imbs, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Set-Constrained Delivery Broadcast: Definition, Abstraction Power, and Computability Limits. In Paolo Bellavista and Vijay K. Garg, editors, *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 7:1–7:10. ACM, 2018. doi:10.1145/3154273.3154296.

9 Damien Imbs and Michel Raynal. Trading off t-Resilience for Efficiency in Asynchronous Byzantine Reliable Broadcast. *Parallel Processing Letters*, 26(4):1–8, 2016. doi:10.1142/S0129626416500171.

10 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. doi:10.1145/357172.357176.

11 Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system, 2008.

12 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2):228–234, 1980. doi:10.1145/322186.322188.

13 Michel Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*. Springer, 2018. doi:10.1007/978-3-319-94141-7.

14 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011. doi:10.1007/978-3-642-24550-3_29.

A Two Multi-shot Signature-free 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 [3] and D. Imbs and M. Raynal [9]. The presentation follows pages 64-71 of [13], 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

In both 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. As a consequence, a process can identify the sender of each message it receives and no Byzantine process can impersonate another process. In practice, this means that Byzantine processes cannot control the underlying communication layer.

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

Algorithm 4 is a 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).

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

```
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, −) received from p_j;
(3)   broadcast ECHO(⟨j, sn⟩, m).

when a message ECHO(⟨j, sn⟩, m) is received from any process do
(4)   if   (ECHO(⟨j, sn⟩, m) received from strictly more than (n+t)/2 different processes)
          ∧ (READY(⟨j, sn⟩, m) not yet broadcast)
(5)      then broadcast READY(⟨j, sn⟩, m)
(6)   end if.

when a message READY(⟨j, sn⟩, m) is received from any process do
(7)   if   (READY(⟨j, sn⟩, m) received from at least (t + 1) different processes)
          ∧ (READY(⟨j, sn⟩, m) not yet broadcast)
(8)      then broadcast READY(⟨j, sn⟩, m)
(9)   end if;
(10)  if    (READY(⟨j, sn⟩, m) received from at least (2t + 1) different processes)
          ∧  (⟨j, sn, m⟩ not yet br_delivered)
(11)     then br_deliver ⟨j, sn, m⟩
(12)  end if.
```

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[1].

- 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$(\langle j, sn\rangle, m)$ from 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 exchanges ensure 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 fromp_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$(\langle j, sn\rangle, m)$ for "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 [13] 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

Algorithm 5 is a 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).

[1] 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 (see the paragraph "Invocation pattern" at the end of Section 2.3).

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

```
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, −) received from p_j;
(3)   broadcast WITNESS(⟨i, sn⟩, m).

when WITNESS(⟨j, sn⟩, m) is received from any process do
(4)   if (WITNESS(⟨j, sn⟩, m) received from at least (n − 2t) different processes)
        ∧ (WITNESS(⟨i, sn⟩, m) not yet broadcast)
(5)     then broadcast WITNESS(⟨j, sn⟩, m)
(6)   end if;
(7)   if (WITNESS(⟨j, sn⟩, m) received from at least (n − t) different processes)
        ∧ (⟨j, sn, m⟩ not yet br_delivered)
(8)     then br_deliver ⟨j, sn, m⟩
(9)   end if.
```

- 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).

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 $t < n/5$.

B Proofs of Algorithms

This appendix contains the complete proofs for the theorems that were not proved in the paper.

B.1 BFIFO-broadcast on top of MBR-broadcast

BFIFO-Validity follows directly from MBR-Validity. BFIFO-Integrity (resp. BFIFO-Termination) follows directly from MBR-Integrity (resp. MBR-Termination), the delivery condition (line 3), and the increase of $fifo\_del_i[j]$ (line 5). The formalization of these proofs are left to the reader. The next (easy) theorem concerns the BFIFO-Order property.

▶ **Theorem 24.** *Algorithm* 1 *satisfies the* BFIFO-Order *property.*

Proof. Suppose a correct process p_i bfifo-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 bfifo-delivers message from any process in the order defined by their sequence numbers, we have $sn < sn'$. It follows then from MBR-Termination-2 that all correct processes br-deliver m and m' with sequence numbers sn and sn', respectively. It follows that any correct process bfifo-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 bfifo-broadcast m before m'. ◀

The computability and messages/time complexities of this algorithm are the ones of the underlying Byzantine reliable broadcast algorithm.

B.2 BSCD-broadcast in action: a Byzantine-tolerant snapshot object

We start by several intermediate definitions and results before the final proof of Theorem 23.

▶ **Lemma 25.** *If a correct process invokes an operation, it returns from its invocation.*

Proof. Let p_i be a correct process that invokes a read or write operation. By the BSCD-Termination-1 property of BSCD-broadcast, it eventually receives a message set containing the message $\langle i, sni, x\rangle$ which it sent at line 1 or 3 of Algorithm 3. As all the statements associated with the bscd-delivery of a message set terminate, it follows that the synchronization boolean $done_i$ is eventually set to `true`. Consequently, p_i returns from the invocation of its operation. ◀

▶ **Definition 26.** *A sequence number array* sna *is an array $sna = [sna[1], \ldots, sna[n]]$ of n sequence numbers, one per process.*

▶ **Remark 27.** *In Algorithm 3, wsn_i is a sequence number array that is used to keep track of p_i's vision of the operations performed so far.*

▶ **Definition 28.** *Let $\leq_{sna}$ be the product order defined on sequence number arrays as:*

$$sna_1 \leq_{sna} sna_2 \iff \forall k \in 1..n,\ sna_1[k] \leq sna_2[k].$$

Let $<_{sna}$ be the relation defined as:

$$sna_1 <_{sna} sna_2 \iff (sna_1 \leq_{sna} sna_2) \wedge (sna_1 \neq sna_2).$$

▶ **Definition 29.** *If p_i is a correct process, let WSN_i be the set of the array values taken by wsn_i at line 7 during an execution, after the processing of message sets by process p_i. Let $WSN = \bigcup_{p_i \text{ correct}} WSN_i$.*

▶ **Lemma 30.** *The order $<_{sna}$ is total and well founded on WSN.*

Proof. Let us first observe that, for any correct process p_i, all values in WSN_i are totally ordered: this comes from wsn_i whose entries can only increase (line 5). Hence, let sna_1 be an array value of WSN_i and sna_2 be an array value of WSN_j with $i \neq j$ and where both p_i and p_j are correct processes.

Let us assume, by contradiction, that $\neg(sna_1 \leq_{sna} sna_2)$ and $\neg(sna_2 \leq_{sna} sna_1)$. Thus there is a k such that $sna_1[k] > sna_2[k]$ and a k' such that $sna_1[k'] < sna_2[k']$. According to lines 4 and 5, there is a message $\langle k, sna_1[k], \text{WRITE}(v)\rangle$ that has been received by p_i when $wsn_i = sna_1$ and not by p_j when $wsn_j = sna_2$. Similarly, there is a message $\langle k', sna_2[k'], \text{WRITE}(v')\rangle$ that has been received by p_j when $wsn_j = sna_2$ and not by p_i when $wsn_i = sna_1$. This situation directly contradicts the BSCD-Order property, from which we conclude that either $sna_1 \leq_{sna} sna_2$ or $sna_2 \leq_{sna} sna_1$. Therefore $<_{sna}$ is a total order.

Since all elements of WSN are vectors of elements of $\mathbb{N}$, they all have a finite number of strictly smaller elements, therefore $<_{sna}$ is well founded on WSN. ◀

▶ **Definition 31.** *Let C be the set of triples $\langle i, sni, x\rangle$ bscd-broadcast by correct processes when invoking an operation (line 1 or 3), where i and sni is the control information provided by the BSCD-broadcast abstraction and x is the value sent, either* SYNC() *or* WRITE(v).

Since each operation invoked by a correct process corresponds to the BSCD-broadcast of a unique message, we will identify the corresponding message triple with the invoked operation.

For any $\langle i, sni, x\rangle, \langle j, snj, y\rangle \in C$, we write $\langle i, sni, x\rangle \prec \langle j, snj, y\rangle$ if p_i returned from its operation that bscd-broadcast $\langle i, sni, x\rangle$ before p_j started its operation that bscd-broadcast $\langle j, snj, y\rangle$.

▶ **Definition 32.** *Let W be the set of triples $\langle i, sni, \text{WRITE}(v)\rangle$ bscd-delivered by correct processes. This set may contain write operations by Byzantine processes.*

▶ **Definition 33.** *Let $O = W \cup C$ be the set of read (or SYNC) and write operations invoked by correct processes and of write operations received by correct processes from Byzantine processes.*

▶ **Remark 34.** *As a direct consequence of BSCD-broadcast properties, the set O contains at most one triplet of the form $\langle i, sni, -\rangle$ for given i and sni.*

▶ **Lemma 35.** $\langle i, sni, \text{WRITE}(v)\rangle \prec \langle i, sni', \text{WRITE}(v')\rangle \implies sni < sni'$.

Proof. $\prec$ only applies to operations by correct processes, therefore p_i is a correct process. If a correct process terminates WRITE(v) before starting WRITE(v'), then it bscd-broadcast WRITE(v) before WRITE(v'), therefore $sni < sni'$. ◀

▶ **Definition 36.** *If sna is a sequence number array, let $W(sna) = \{\langle i, sni, \text{WRITE}(v)\rangle \in W : sni \leqslant sna[i]\}$ be the set of write operations included in the time barrier defined by sna.*

▶ **Definition 37.** *If τ is a time at which a correct process p_i executes line 7, then:*
- *let $\text{wsn}_i(\tau)$ be the value of wsn_i when p_i executes line 7 at time τ*
- *let $U_i(\tau)$ be the union of all the message sets bscd-delivered by p_i until τ*

▶ **Lemma 38.** *$U_i(\tau) \cap W = W(\text{wsn}_i(\tau))$. As a consequence:*

$$U_i(\tau) \cap W \subseteq U_j(\tau') \cap W \iff \text{wsn}_i(\tau) \leqslant_{sna} \text{wsn}_j(\tau'),$$
$$U_i(\tau) \cap W \subsetneq U_j(\tau') \cap W \iff \text{wsn}_i(\tau) <_{sna} \text{wsn}_j(\tau').$$

Proof. Let $\langle j, snj, \text{WRITE}(v)\rangle$ be a message of W. If $\langle j, snj, \text{WRITE}(v)\rangle \in U_i(\tau)$, then when p_i executes line 7 at time τ, it has already bscd-delivered and processed $\langle j, snj, \text{WRITE}(v)\rangle$, therefore $\text{wsn}_i(\tau)[j] \geq snj$, therefore $\langle j, snj, \text{WRITE}(v)\rangle \in W(\text{wsn}_i(\tau))$. Conversely, if $\langle j, snj, \text{WRITE}(v)\rangle \in W(\text{wsn}_i(\tau))$, then at time τ p_i has already bscd-delivered and processe a message $\langle j, snj', \text{WRITE}(v')\rangle$ with $snj' \geq snj$. Since BSCD-broadcast preserves sequence number ordering, p_i has already bscd-delivered $\langle j, snj, \text{WRITE}(v)\rangle$, therefore $\langle j, snj, \text{WRITE}(v)\rangle \in U_i(\tau)$.

The two equivalency relations follow directly from replacing $U_i(\tau) \cap W$ and $U_j(\tau') \cap W$ respectively by $W(\text{wsn}_i(\tau))$ and $W(\text{wsn}_j(\tau'))$ and applying the definitions. ◀

▶ **Definition 39.** *If $op = \langle i, sni, x\rangle \in O$ then let* sna(op), witness(op) *and $\tau(op)$ be defined as follows:*
- *if $x =$*SYNC(), *then by definition of O, p_i is a correct process. Let:*
  - sna(op) *be the value of wsn_i when p_i returns $reg_i[1 \ldots n]$ (line 2),*
  - witness(op) $= p_i$,
  - *$\tau(op)$ be the time at which p_i executes line 7 for the last time before returning at line 2.*
- *if $x =$*WRITE(v) *then let:*
  - sna(op) *be the smallest sequence number array observed by a correct process and that has registered the write operation op, which exists because $<_{sna}$ is a well-founded total order on WSN (by Lemma 30):*
    $$\text{sna}(op) = \min\{sna \in WSN : sna[i] \geqslant sni\},$$

- $\tau(op)$ *be the smallest time at which a correct process* p_j *executed line 7 with* $wsn_j = \text{sna}(op)$,
- $\text{witness}(op) = p_j$ *be the correct process that executed line 7 with* $wsn_j = \text{sna}(op)$ *at* $\tau(op)$.

In both cases, the following relations hold:

$$\begin{aligned} \text{sna}(op) &= \text{wsn}_{\text{witness}(op)}(\tau(op)), \\ op &\in U_{\text{witness}(op)}(\tau(op)). \end{aligned}$$

And if p_i *is a correct process that invoked* op *at* $\tau_{invoke}(op)$ *and returned at* $\tau_{return}(op)$ *then:*

$$\tau_{invoke}(op) < \tau(op) < \tau_{return}(op).$$

▶ **Lemma 40.** *Let* op *and* op' *be two distinct operations such that* $op \prec op'$. *We have* $\text{sna}(op) \leqslant \text{sna}(op')$. *Moreover, if* $op' \in W$ *then* $\text{sna}(op) < \text{sna}(op')$.

Proof. op and op' are invoked by correct processes. We have $\tau(op) < \tau_{return}(op) < \tau_{invoke}(op') < \tau(op')$.

By BSCD-order, we have:

$$U_{\text{witness}(op)}(\tau(op)) \subseteq U_{\text{witness}(op')}(\tau(op')) \quad \vee \quad U_{\text{witness}(op')}(\tau(op')) \subseteq U_{\text{witness}(op)}(\tau(op)).$$

However op' was bscd-broadcast after $\tau(op)$ therefore $\text{witness}(op)$ cannot have bscd-delivered op' at time $\tau(op)$. Therefore: $op' \in U_{\text{witness}(op')}(\tau(op'))\backslash U_{\text{witness}(op)}(\tau(op))$.

Therefore we are in the case: $U_{\text{witness}(op)}(\tau(op)) \subset U_{\text{witness}(op')}(\tau(op'))$, from which we deduce:

$$\begin{aligned} U_{\text{witness}(op)}(\tau(op)) \cap W &\subseteq U_{\text{witness}(op')}(\tau(op')) \cap W, \\ \text{wsn}_{\text{witness}(op)}(\tau(op)) &\leqslant \text{wsn}_{\text{witness}(op')}(\tau(op')), \\ \text{sna}(op) &\leqslant \text{sna}(op'). \end{aligned}$$

Moreover, if $op' \in W$ then the inclusion and inequalities become strict and we have the desired results. ◀

▶ **Definition 41.** *Let* $op, op' \in O$ *be two distinct operations. We define* $\to$ *as* $op \to op'$ *if either one of the following holds:*

- (A) $op \prec op'$,
- (B) $\text{sna}(op) < \text{sna}(op')$,
- (C) $(\text{sna}(op) = \text{sna}(op')) \wedge (op \in W) \wedge (op' \notin W)$.

Let $\to^*$ *be the reflexive transitive closure of* $\to$.

▶ **Lemma 42.** $\to^*$ *is a partial order on* C.

Proof. $\to^*$ is reflexive transitive by construction. Let us prove antisymetry:

Suppose we have $op \neq op'$ such that $op \to^* op'$ and $op' \to^* op$. By definition of $\to^*$ this means there are $op_0, op_1, op_2, \ldots, op_m$ for an $m > 1$ such that $op_0 = op_m = op$, $op_k = op'$ for some $k \in [1, m-1]$, and $op_i \to op_{i+1}$ for all $i < m$.

By the definition and Lemma 40, we have that $\forall i < m, \text{sna}(op_i) \leqslant \text{sna}(op_{i+1})$.

Moreover $\text{sna}(op_0) = \text{sna}(op_m)$, and $\leqslant$ on sequence number arrays is antisymmeteric, thus we have: $\forall i < m, \text{sna}(op_i) = \text{sna}(op_0)$, which eliminates case (B) for all i.

We have two cases:

- If for any $i < m$, $op_i \notin W$, then we are necessarily in case (A): $op_i \prec op_{i+1}$. Since $\text{sna}(op_i) = \text{sna}(op_{i+1})$, by the contrapositive of Lemma 40, we have $op_{i+1} \notin W$.
  By applying this same reasonning recursively, we have that $\forall i < m, op_i \prec op_{i+1}$. As $\prec$ is included in the total order of time, it is a partial order, therefore antisymmetric. We have a contradiction.
- Otherwise, $\forall i < m, op_i \in W$, which eliminates case (C). By the contrapositive of Lemma 40, since $\text{sna}(op_i) = \text{sna}(op_{i+1})$, we have $\forall i, op_i \not\prec op_{i+1}$. All three cases for $op_i \to op_{i+1}$ are impossible, we have a contradiction.

In both cases we have shown a contradiction, therefore $\to^*$ is antisymmetric. ◀

Final proof of Theorem 23

Proof. Let $\to_{op}$ be a topological sort of $\to^*$. $\to_{op}$ is a total order that extends $\to^*$. $\to_{op}$ includes the process order $\prec$ therefore it is real-time compliant.

Let us consider a snapshot operation $op = \langle i, sni, \text{SYNC}() \rangle$ by a correct process p_i, and let us show that the value read by p_i in each register is the last value written to that register before op according to $\to_{op}$.

Let us consider a register k and let $snk = \text{sna}(op)[k]$. According to line 3, the value returned by op for register k is a value v such that $op_k = \langle k, snk, \text{WRITE}(v) \rangle$ is the last write operation on k known by p_i when the snapshot operation returns. By definition of $\text{sna}(op)$ we have $\text{sna}(op_k) \leqslant \text{sna}(op)$, moreover op_k is a write and op is a read, therefore $op_k \to_{op} op$. Moreover, for any different write operation $op'_k = \langle k, \text{snk}', \text{WRITE}(v') \rangle$ on register k, we have $snk \neq snk'$. If $snk' < snk$ then $op'_k \to_{op} op_k$. Otherwise $snk' > snk$, in which case $\text{sna}(op'_k) > \text{sna}(op)$, therefore $op \to_{op} op'_k$. In both cases, v is the last value written on register k before the snapshot operation op according to $\to_{op}$.

We have thus shown that Algorithm 3 builds a linearizable single-writer multi-reader Byzantine tolerant snapshot object. ◀